Sistemas Informáticos

Curso 2011 - 2012

Un Sistema de Tutorización Inteligente para el Aprendizaje de Estructuras de Datos Arborescentes

Kangmin Lee Song

Beatriz Martín San Juan

Christian Morales Peña

Dirigido por Rafael del Vado Vírseda

Departamento de Sistemas Informáticos y Computación

Facultad de Informática

Universidad Complutense de Madrid

Agradecimientos

Nos gustaría agradecer tanto las ayudas y las opiniones como las críticas constructivas recibidas a lo largo del desarrollo de este proyecto.

Primero de todo, agradecer a nuestro tutor de proyecto, Rafael del Vado Vírseda, su confianza depositada en nosotros para realizar este trabajo.

A nuestros amigos y compañeros que nos han apoyado todo este tiempo de distintas maneras y que, en algunos casos, han participado activamente opinando sobre la utilizad de la herramienta, dando ideas de cómo hacer ciertas partes más atractivas. Decirles que sus comentarios nos han servido para llevar una cabo una aplicación que creemos que puede ser utilizada por todo tipo de usuarios. A todos muchas gracias.

Página de autorización

Por el presente texto se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código fuente generado y/o el prototipo desarrollado.

Madrid, 24 de Junio de 2012

Fdo.: Rafael Del Vado

Fdo.: Kangmin Lee Song

Fdo.: Beatriz Martín San Juan

Fdo.: Christian Morales Peña

Índice

RESUMEN DEL PROYECTO

En castellano

V-Tree es un proyecto que recoge las ideas presentadas en trabajos anteriores de Sistemas Informáticos como Vedya [1], Vedya-Test [2] y las ha orientado para crear una herramienta educativa que ofrece un entorno de aprendizaje de estructuras de datos arborescentes AVLs y Rojinegros. Este aprendizaje estará guiado por un Tutor inteligente, que controlará los avances del alumno y le guiará a lo largo del mismo.

La herramienta **V-Tree** pone a disposición dos modos de uso. Un modo estudiante, en el que el alumno podrá construir sus propias estructuras arborescentes haciendo uso de un panel gráfico animado o realizar test de evaluación para mostrar el correcto entendimiento de estas estructuras; y un modo profesor, en el que se podrá gestionar la batería de preguntas y tests ya existentes, de forma que pueda crear cuestiones y combinarlas con otras para crear nuevos tests.

En inglés

V-Tree collects the ideas presented in “Computer Systems” earlier works such as Vedya and Vedya-Test. It has used them to create a new educational tool which will provide an useful learning enviroment for AVL and Red-Black trees Data Structures. The learning will be guided by an Intelligent Tutor who will control the student’s progress .

V-Tree tool has two differents ways of being used. Students will build their own tree structures using an animated graphics or tests their knwodledges. Teachers will write new questions, manage databases and handle tests.

1. Introducción

Los Sistemas de Aprendizaje Interactivos (SAIs) cada vez ocupan un papel más importante en el mundo de la docencia. Son modelos innovadores de enseñanza que se basan (sostiene) en el aprendizaje dentro de estudios muy específicos bajo el propósito de mejorar y facilitar la aprehensión y comprensión por parte del alumno.

De entre todos los beneficios presentes en estos sistemas resaltan la interoperabilidad y la reusabilidad desde un punto de vista tecnológico y educativo. La posibilidad de crear un recurso educativo que pueda ser especializado en diversos campos de estudio y accesible desde diversas localizaciones sin importar dónde fue creado, aporta una nueva motivación para el desarrollo y utilización de estos sistemas.

Existen numerosas herramientas comercializadas y de libre acceso implementadas bajo esta iniciativa. Hacia esta dirección se ha orientado el desarrollo de nuestra aplicación **V-Tree**. Dentro de la organización de los Sistemas de Aprendizaje Interactivos, **V-Tree** se encuentra enfocado hacia los Sistemas de Tutorización Inteligentes. Esta rama derivada de los SAIs añade a todo lo descrito anteriormente el seguimiento del alumno por parte de un **Tutor Inteligente,** que será el encargado de guiar al estudiante a lo largo del proceso de aprendizaje.

Los precedentes de este proyecto se hallan en las herramientas anteriormente desarrolladas por la Facultad de Informática UCM, *Vedya* y *VedyaTest*. La primera especializada en la enseñanza de Estructuras de Datos y Métodos Algorítmicos; y la segunda es un primer paso para la generación de tests de evaluación sobre los temas anteriores.

Los objetivos principales de **V-Tree** han sido aunar estas dos ideas enfocándolas hacia el aprendizaje de Estructuras de Datos Arborescentes e innovar en la generación y edición de tests. Para cumplirlos se han seguido las directrices de los Sistemas de Tutorización Inteligente descritos en el siguiente apartado.

2. Sistemas de Tutorización Inteligentes

Los STIs se desarrollaron con la intención de proporcionar un marco prometedor de aplicaciones que pudiera servir como campo de experimentación para educadores, psicólogos y teóricos cognitivos, haciendo uso de las técnicas de la Inteligencia Artificial. A esto hay que añadir el interés que ha despertado el uso de los ordenadores en la educación.

Los STIs tienen unos objetivos de aprendizaje claramente definidos. Los desarrolladores de este tipo de sistemas adoptan una perspectiva objetivista, es decir, establecen que el conocimiento se puede estructurar de forma completa y correcta en términos de entidades, propiedades y relaciones. Partiendo de esta idea, esta perspectiva sostiene que el pensamiento racional consiste en la manipulación de símbolos abstractos vistos como representantes de la realidad.

Con este tipo de desarrollos se ha intentado probar que mejoran la velocidad y la calidad del aprendizaje de los alumnos. Para ello hacen uso de métodos tradicionales de aprendizaje, enseñanza y evaluación. Los sistemas WEST y SOPHIE [Burton y Brown 1976, 1975] destacan entre los primeros desarrollos de este tipo.

Los STIs intentan manejar un método de enseñanza y aprendizaje ejemplificado por la interacción humana de tutoría uno a uno. Más concretamente, se basan en la interacción sistema-humano, ejercicio-práctica para llevar a la práctica su método.

Esta metodología es ampliamente aceptada tanto por la comunidad educacional como por nuestra cultura natural. La tutoría uno a uno permite un aprendizaje individualizado y conduce a mejores resultados que otros métodos. Como desventajas, encontramos la inadecuación de las técnicas de evaluación de los resultados del aprendizaje.

El esquema básico modular de un STI es el siguiente:

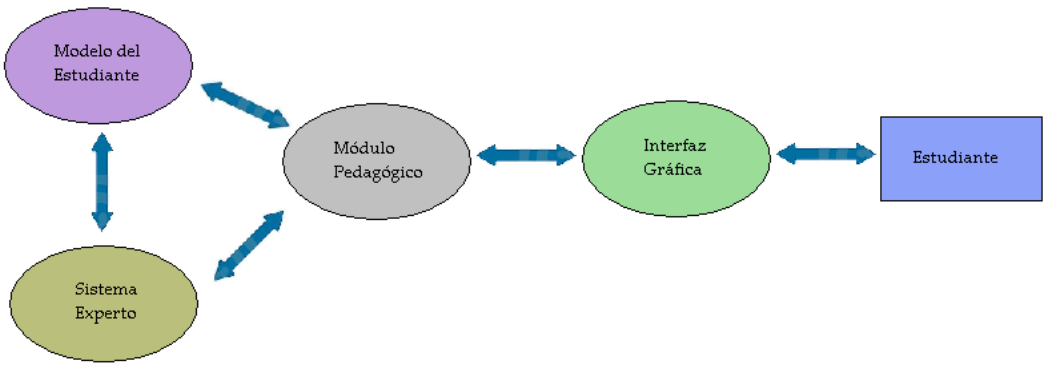

Figura 1. Esquema básico modular de un STI

Estos sistemas se caracterizan por una filosofía que incluye un control del tutor y un formato de tarea de respuesta corta. Los estudiantes aprenden trabajando en series de cuestiones relativamente breves y el sistema juega exclusivamente el papel de experto en la tarea, controlando la selección de problemas, mientras que el estudiante es el responsable de resolverlos.

Los componentes básicos de un STI son los mostrados en el esquema anterior, los cuales pasamos a describir brevemente:

- **Sistema experto o conocimiento del dominio:** contiene una representación del conocimiento específico del área en cuestión.

- **Modelo del estudiante**: el conocimiento del estudiante se pone en relación con el conocimiento del dominio. Podemos decir que el conocimiento del alumno se expande hasta coincidir con el conocimiento del dominio y que para ello se usan diversos métodos para comparar dichas diferencias entre conocimientos. A este modelo se le conoce como modelo *overlay o superpuesto*.

- **Módulo pedagógico**: la didáctica comprende todas aquellas actividades que tienen efecto directo en el estudiante. Se puede hacer una descomposición en los siguientes elementos:

  - **Acciones didácticas**: son las operaciones básicas realizadas en el proceso. Dicho proceso se organiza en forma de planes, que explican de forma natural la organización de currículos y otras actividades didácticas. Cada acción puede provocar una serie de expectativas didácticas sujetas a un proceso de inferencia.

  - **Contexto estratégico**: es donde se realizan las acciones didácticas, conducido mediante objetivos. Los objetivos se pueden plasmar en la ejecución de planes y en la detección de oportunidades de intervención a partir de los datos observados.

  - **Restricciones**: encontramos restricciones provenientes del dominio, didácticas y generadas por el diagnóstico.

  - **Recursos**: distinguimos los recursos locales y los globales.

- **Interfaz con el alumno**: es el vínculo final de un proceso más profundo que comprende todo lo relativo a la comunicación del conocimiento con el alumno. El problema de la comunicación del conocimiento es esencial en el proceso de aprendizaje. El funcionamiento debería ser tal que la interfaz adoptase la representación más adecuada en función del estado de conocimiento.

¿Cómo entran en relación estos componentes?

Mediante la interfaz gráfica, el STI transmite al alumno los conocimientos representados en el modelo de conocimientos del dominio, siguiendo las directrices propuestas en el módulo pedagógico.

Cada vez que el estudiante comete un error, el sistema diagnostica el problema e intenta solucionarlo con un consejo muy detallado acerca de cómo habría actuado él (el sistema experto) para resolverlo.

Para que este esquema de funcionamiento sea factible se crea un marco de interacciones entre los distintos módulos. El control lo ejerce el módulo pedagógico en función del conocimiento operativo existente.

En cuanto a la representación, debemos distinguir entre representación interna y representación externa, refiriéndose la primera a los mecanismos de representación e inferencia del conocimiento, por lo que el conocimiento del dominio reside aquí, y refiriéndose la segunda a la forma en la que la interfaz se presenta ante el alumno.

La representación externa ha sufrido muchas variaciones debido a los avances en las interfaces. Estos avances tienen un impacto significativo en la calidad del aprendizaje, por lo que no debemos descuidar en ningún momento este apartado. La representación externa juega un papel crítico durante el aprendizaje.

La mayor virtud de los STIs es su capacidad para generar realimentación altamente detallada acerca de la resolución de un problema, microtutorizar. Una rica realimentación conduce a diagnósticos más precisos de errores y, por tanto, a un aprendizaje más rápido.

Ofrecen un mayor control del aprendizaje por parte del tutor. En la mayoría de los casos es el software el que selecciona la siguiente tarea o problema y decide cuando el estudiante necesita soporte y realimentación en la resolución de un problema.

Estos sistemas son congruentes con la práctica de clases en dos sentidos. En primer lugar, se plantean objetivos ya incluidos en la tradición curricular. Por otro lado, adoptan un método popular de enseñanza y aprendizaje al combinar la disertación con la práctica de ejercicios. Los profesores, por tanto, no encuentran problema en incorporar estas herramientas a sus clases.

Por el contrario, los STIs no se desarrollaron con éxito en áreas no tan bien conocidas o comprendidas como pueden ser las Ciencias Sociales y la Historia. En aquellos problemas donde no se pueden construir sistemas expertos un STI pierde su potencial.

Los componentes pedagógicos no llegan a contemplar procesos más elaborados de aprendizaje relacionados con el significado de los conceptos y habilidades adquiridos. La ciencia cognitiva aún no ha progresado hasta el punto de ofrecer un buen análisis de las tareas de un experto pedagogo.

La libertad del estudiante está siempre circunscrita ya que los STIs ofrecen limitadas elecciones y es el software el que selecciona la siguiente tarea o problema.

Para mejorar las capacidades pedagógicas de un STI es preciso enriquecer la base de conocimiento pedagógica, mejorando entre otras cosas la base de reglas que infiere cuándo y cómo instruir a los estudiantes, e implementar un mejor método de enseñanza y aprendizaje. Los STIs están ceñidos a un único modelo de enseñanza y aprendizaje mientras que un tutor experto puede adoptar diferentes métodos. Carecen de capacidad para tutorizar de forma flexible, adoptar diferentes métodos de enseñanza cuando resulte apropiado y permitir a los estudiantes utilizar distintos tipos de aprendizaje

El método de enseñanza ejercicio-práctica de los STIs parece más adecuado para sintonizar o refinar conocimientos ya existentes que para el aprendizaje conceptual de piezas substanciales de nuevo conocimiento. Esto refleja las limitaciones de la ciencia cognitiva que subyace en los STIs.

Las nuevas investigaciones intentaron mejorar el método de tutoría uno a uno, bien investigando diferentes métodos de aprendizaje y enseñanza o bien aumentando el rango de objetivos y resultados del aprendizaje y , al mismo tiempo, intentando desarrollar software educacional en áreas más diversas.

El aprendizaje derivado de un STI puede mejorar tan sólo con hacerlo más amigable mejorando las interfaces gráficas. De este modo, la comunicación entre alumno y tutor se vuelve más simple y ofrece canales más diversos y las explicaciones visuales resultan más fáciles de entender y más entretenidas. Los modelos contenidos en los STIs también han mejorado ya que la ciencia cognitiva ha continuado facilitando mejores modelos cognitivos y análisis más detallados de tareas para formas cada vez más sofisticadas de razonamiento y resolución de problemas.

Como hemos mencionado previamente, los límites de los STIs son los mismos de la IA, de quien heredan sus métodos. No son capaces de capturar todo el conocimiento experto práctico, y están construidos para desempeñar una tarea, no para enseñarla. En cualquier habilidad cognitiva compleja, la capacidad de memorizar y aplicar un conjunto de reglas es menos valiosa educacionalmente que comprender el significado y la génesis de las decisiones de diagnóstico. Los sistemas expertos tradicionales tienden pues a impartir al estudiante conocimiento poco profundo.

Entornos de Aprendizaje Interactivo y Micromundos

Estos sistemas surgieron como respuesta a las limitaciones del enfoque adoptado en muchos STIs en la etapa previa.

La teoría de aprendizaje utilizada era el *constructivismo*. El método de aprendizaje se denomina *basado en indagación*. El constructivismo pone énfasis en los procesos de estructuración activa del mundo y sostiene que existen múltiples significados o perspectivas para cualquier evento o concepto.

La inteligencia de los Entornos Interactivos de Aprendizaje (abreviadamente, EAI) se distribuye entre un conjunto de herramientas en lugar de centrarse en un tutor. Estas herramientas permiten a los estudiantes investigar y aprender libremente, sin un control

externo. Esta libertad aporta beneficios prácticos ya que los EAIs no son tan intensivos en conocimiento como los STIs. Los EAIs proporcionan una representación de los temas que el estudiante debe investigar, pero no tienen por qué conocer "todas las respuestas correctas", ni deben incluir modelos de la cognición del estudiante, ni deben tomar complejas decisiones pedagógicas. Por otra parte, quizá se sobrevalora la capacidad de los alumnos para descubrir ideas interesantes o juzgar qué tipo de conocimiento deben construir, causando que, en ocasiones, el estudiante acabé entre cuestiones sin interés.

Los *Micromundos* son un tipo particular de EIAs que siguen un método basado en la indagación y que suponen un cambio en los objetivos del aprendizaje. En primer lugar, se siguen considerando importante el aprendizaje de la caracterización de patrones de relaciones entre objetos y propiedades que definen el mundo. En segundo lugar, muchos *Micromundos* alientan a adquirir habilidades de indagación.

Métodos y objetivos del aprendizaje

Los *Micromundos* suponen un concepto de "herramientas educacionales" y aprendizaje basado en la indagación. Las principales habilidades de indagación que se persiguen son:

- Proponer cuestiones
- Proponer problemas específicos
- Generar conjeturas o hipótesis
- Recopilar observaciones que conduzcan a cuestiones o hipótesis
- Confirmar o invalidar hipótesis
- Refinar hipótesis
- Explicar o probar una hipótesis

La experimentación ha manifestado que los STIs dan resultados deseables para un sistema educacional como puede ser: aprendizajes de conceptos inesperados, aprendizaje de conceptos externos al *Micromundo* y aprendizaje genérico de la propia materia.

Aunque los EIAs son mucho más diversos que los STIs. Sin embargo, comparten varios principios que contrastan fuertemente con los puntos de vista implícitos en los STIs.

- Construcción frente a Instrucción
- Control del estudiante frente a control del autor
- Individualización determinada por el estudiante, no por el tutor.
- Rica realimentación generada por la interacción del estudiante con el entorno de aprendizaje, no con el tutor.

Logros y limitaciones

- Uso del conocimiento por delegación y reparto de papeles frente a tutoría estricta: En lugar de requerir que los estudiantes estudien con problemas concretos, permiten al estudiante escoger cualquier problema. La práctica de los estudios se focaliza en automatizar muchas de las habilidades menos importantes, delegando éstas al sistema.

- Desacoplamiento entre el experto EAI y el aprendizaje del estudiante: Proporcionan entornos que permiten al estudiante construir su propio conocimiento, pudiéndose desarrollas incluso en áreas en que éstos no lo han sido.
- Problemas cognitivos: Gran cantidad del comportamiento del estudiante puede resultar irrelevante, además existen maneras más rápidas de aprender unas habilidades específicas, sobre todo cuando están claramente definidas.
- Problemas de evaluación (medida de los resultados del aprendizaje): Resulta muy complejo definir los objetivos del aprendizaje, operacionalizándolos en instrumentos objetivos, y aplicarlos para medir la eficacia de los Micromundos en promover dichos objetivos.

Sistemas Hipermedia

Se basa en presentar los contenidos de forma no lineal y estructurada en una red de nodos y enlaces (hipertexto), permitiendo al usuario navegar a través de la información. La forma de introducir y memorizar la información es asociativa.

En un sistema hipermedia, y a diferencia de los hipertextos convencionales, la información contenida en los nodos es multimedia.

En la siguiente figura se muestra una arquitectura general de los sistemas hipermedia. En el primer nivel está el usuario y el autor. El primero puede navegar por la información y el segundo lo actualiza con las herramientas de mantenimiento. El software está separado para estas funcionalidades en el módulo del alumno y en el módulo del profesor. Además, dada la facilidad de perderse por la información, se utiliza el navegador, el cual ayuda al usuario a localizar la información mediante una representación gráfica de la red. También hay métodos de búsqueda que te llevan directamente a un nodo u otro dependiendo de la información que se solicite.

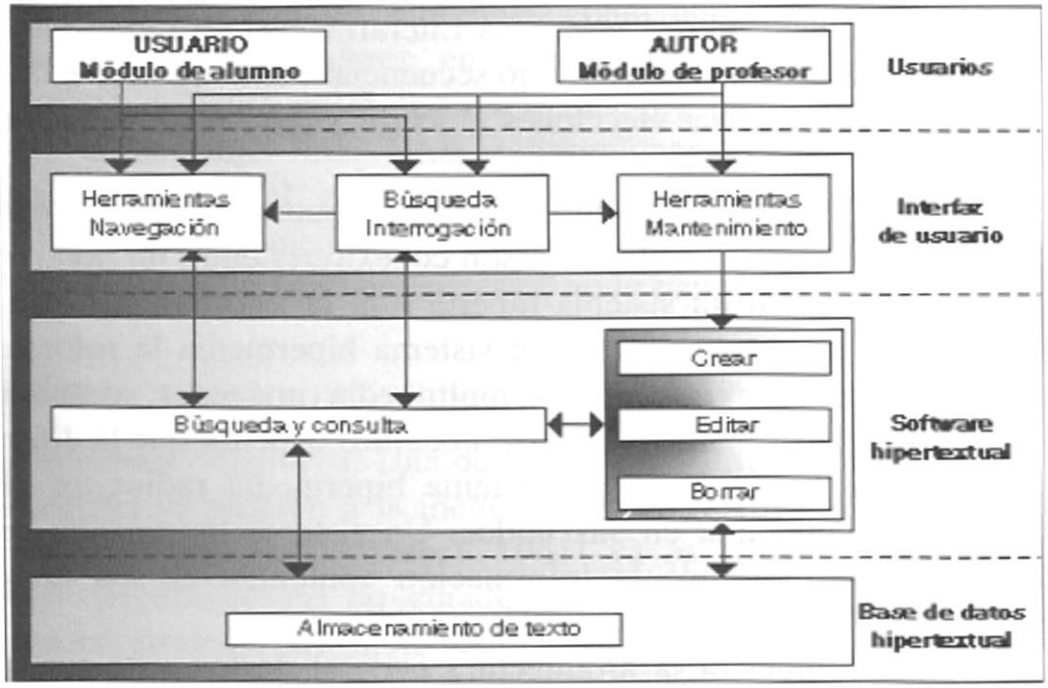

Figura 2. Arquitectura general de los sistemas Hipermedia

Entre las principales motivaciones para el desarrollo de estos sistemas están:

- Proporcionan una alta interactividad.
- Grandes posibilidades para organizar y representar la información de forma no lineal.
- Es una buena manera de representar el conocimiento humano.
- Permiten la adquisición del conocimiento mediante formas diversas.

Quizá lo más destacado sea que la propuesta de hipertexto se ha convertido en un campo cada vez más rico de aplicaciones hipermedia, por ejemplo, mediante el desarrollo de sistemas educativos interactivos con técnicas de inteligencia artificial.

Los sistemas hipermedia han tenido gran aceptación entre los desarrolladores de los sistemas de aprendizaje por ordenador. Estos sistemas se presentan, además, como un marco adecuado donde englobar el proceso de aprendizaje. Estas son algunas de las razones para poder afirmar esto:

- Permiten la adquisición del conocimiento de diversas formas.
- Ayudan al alumno a comprender el significado de lo aprendido.

En su contra, existe una serie de problemas que se han asociado a los hipertextos:

- Desorientación del usuario en la navegación.
- El grado de libertad en el uso de la información por parte de un alumno no debe interferir en el uso que otros alumnos hagan.

Con el fin de solventar estas limitaciones se introdujeron capacidades de adaptación, de modo que se pudiera guiar al alumno en su navegación por los contenidos.

Una de las principales limitaciones está relacionada con la propia estructura de los documentos, y es que no se puede separar contenido de representación. Esto obliga, por ejemplo, a que aplicaciones de IA en la web se centren en aplicar técnicas de filtrado de información para extraer información relevante de las webs, condiciona a que un mismo contenido no se pueda representar en tantos formatos como se desee, que un programa que toma datos dependa del formato de la página y a sus posibles cambios, etc.

Estas limitaciones están condicionadas por el uso del lenguaje HTML (HyperText Markup Language) el cual está aprobado como un estándar por la industria.

Finalmente, otra cuestión asociada es la posibilidad de personalizar sus contenidos en función de las características del usuario. Los sistemas de hipermedia adaptativa permiten filtrar o resaltar la información en función de los intereses del usuario.

Aprendizaje Colaborativo: Sistemas CSCL

Estos sistemas sirven de apoyo al trabajo/aprendizaje competitivo y colaborativo. En líneas generales es un entorno privado para cada estudiante y otro público donde colaboran. Cada uno tienen asistente que le aconseja las acciones que debe llevar a cabo para mejorar la cooperación. Estas recomendaciones se construyen a partir de las comparaciones de sus acciones en el espacio privado y en el público, en base a unas reglas predefinidas.

Los problemas fundamentales que se intenta solventar con esta propuesta son cómo monitorizar al alumno y decidir el comportamiento del asistente. A este respecto, las cuestiones abiertas son siempre, construir un modelo correcto del usuario y de su interacción, qué datos coger y decidir en qué momentos y de qué forma debe interrumpir el asistente al estudiante.

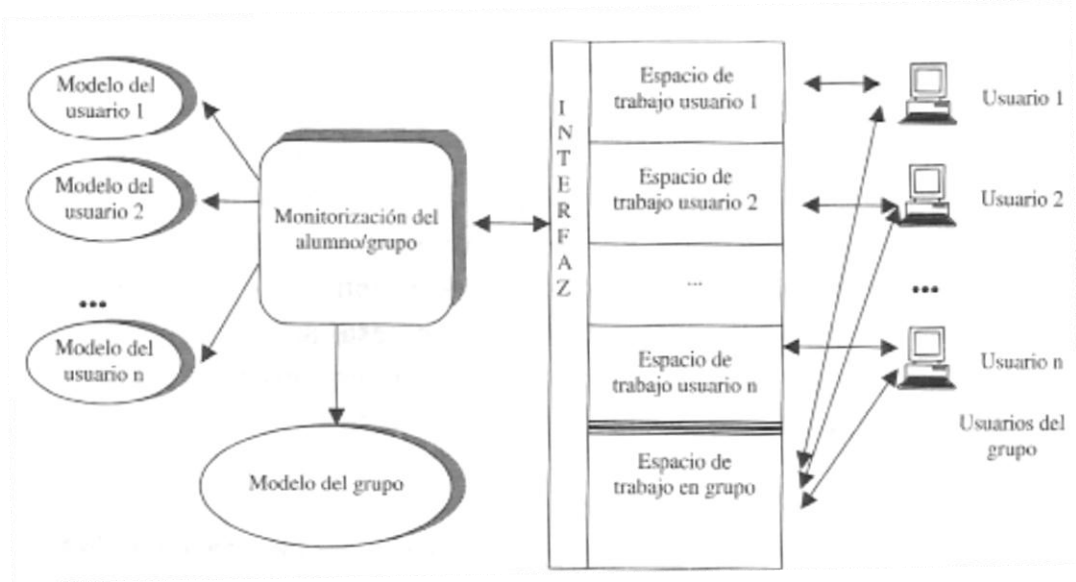

Figura 3. Esquema general de un sistemas CSCL

A diferencia del resto de aplicaciones, es necesario definir un modelo de grupo. Asimismo, cobra especial importancia la correcta monitorización del alumno.

Se pueden distinguir cuatro fases fundamentales en todo sistema de apoyo a la colaboración:

- Recogida de datos.
- Seleccionar indicadores para representar el estado actual de la interacción.
- Diagnóstico de la interacción, comprobando el estado actual de la interacción con un modelo ideal de la misma.
- Sugerencia de acciones a tomar, si el estado de interacción difiere del estado ideal.

3. Un Sistema de Tutorización Inteligente para el Aprendizaje de Estructuras de Datos Arborescentes

3.1. Motivación y objetivos

3.1.1. Motivación

Las estructuras de datos son formas de almacenar y organizar conjuntos de datos para que puedan ser usados eficientemente. Son parte muy importante en la formación de los estudiantes de ingeniería ya que nos permiten desarrollar hábitos correctos de programación, estructurando la mente y ampliando los conocimientos y destrezas innatos en el alumno.

A través del entendimiento de las operaciones y los comportamientos de las diferentes estructuras, se potenciará su correcta utilización y rendimiento en la implementación de algoritmos correctos, consistentes y eficientes.

En este sentido, un papel muy importante lo juega el equipo docente encargado de inculcar estas disciplinas a los estudiantes, por lo que cualquier facilidad que se pueda ofrecer a su actividad, repercutirá directamente en la calidad de la enseñanza y el aprendizaje.

Por otro lado, la oportunidad por parte de los alumnos de poseer una aplicación que aúne práctica y teoría de uso personal, abre un nuevo horizonte en la asimilación de conocimientos.

Por este motivo se ha diseñado **V-Tree**, una herramienta que pretende guiar al alumno en el aprendizaje interactivo de estructuras de datos arborescentes a través de un sistema gráfico de representación de dichas estructuras y un entorno de evaluación en forma de test. Al mismo tiempo se facilitará la labor del profesor ofreciendo distintos mecanismos para la elaboración de material, pudiéndose adaptar a las necesidades de sus estudiantes.

3.1.2. Objetivos

V-Tree tiene como objetivo principal ofrecer un sistema de aprendizaje eficaz y de fácil uso de dos tipos estructuras de datos arborescentes, árboles AVL y árboles rojinegros, así como sentar las bases para futuras ampliaciones del repertorio básico de esta clase de estructuras.

Con dichos objetivos presentes, surgen de manera natural dos modos distintos de utilizar la herramienta: desde el punto de vista práctico y desde el teórico.

Gracias a la primera opción, el alumno podrá acceder a un entorno gráfico de calidad con capacidad para simular por completo el comportamiento de los árboles, poniendo a disposición del usuario un conjunto de acciones a realizar sobre los mismos. Esta sección se ha desarrollado, en la medida de lo posible, intentando mostrar un entorno simple e intuitivo.

Cada acción tendrá como consecuencia una acción gráfica, que realizará los cambios pertinentes en la estructura siguiendo los algoritmos clásicos de inserción, eliminación, rotación y cambio de color. Las animaciones muestran paso a paso el comportamiento de los algoritmos, mostrando las rotaciones a las que da lugar, ayudando a fijar las ideas más importantes. Sin duda esta es una de las partes más interesantes del proyecto.

En la opción teórica se creará una bifurcación; si el usuario se identifica como alumno, éste podrá evaluar sus conocimientos sobre las estructuras de datos mediante los cuestionarios que proporcionará la herramienta. Si se identifica un profesor, se pretenderá facilitar la labor docente. Entre otras actividades podrá crear y modificar sus propias preguntas tipo test, pudiendo utilizar además imágenes capturadas de árboles ya diseñados en la herramienta para dar mayor consistencia y claridad a las preguntas.

3.2. Diseño e Implementación

3.2.1. Diseño

3.2.1.1. Diagramas de clases

A continuación, presentamos las clases principales de la herramienta **V-Tree** mostrando su papel en la implementación

3.2.1.1.1. Estructuras arborescentes

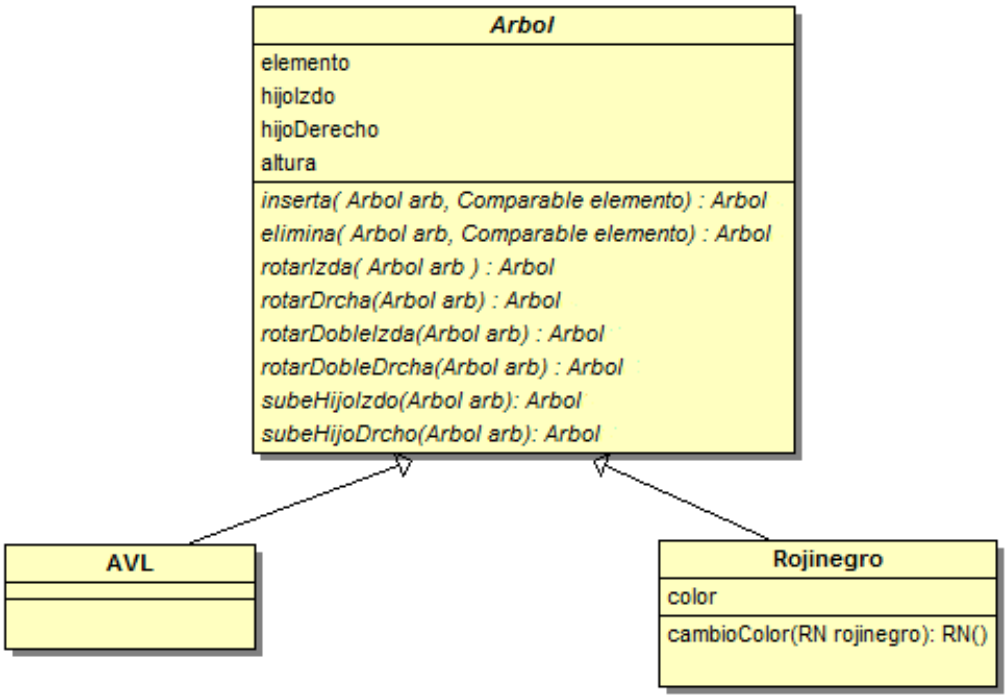

Figura 4. Diagrama del paquete Estructuras

En el paquete *Estructuras* encontramos la clase Árbol, que aporta la funcionalidad básica de los árboles binarios de búsqueda. Tanto los árboles AVL como los rojinegros son árboles binarios de búsqueda, por lo que esta funcionalidad es similar para ambas estructuras. Dentro de este paquete están además las clases "AVL" y "Rojinegro", que aumentan la funcionalidad añadiendo cada una de sus propias restricciones oportunas para las condiciones de equilibrio.

3.2.1.1.2. Gráficos

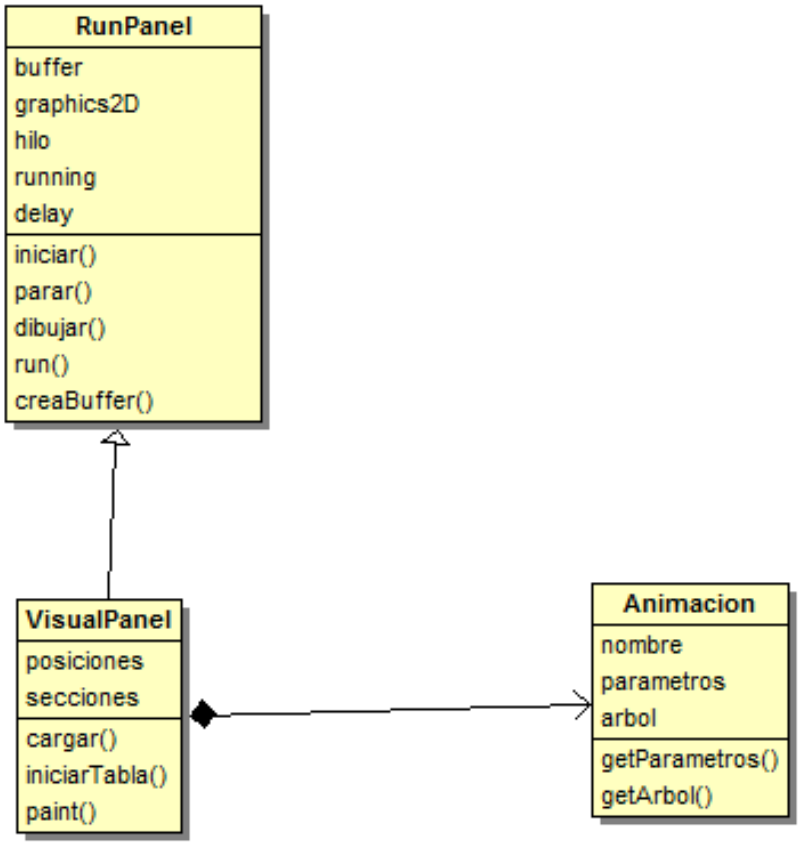

Figura 5. Diagrama del paquete Gráficos

El paquete *Gráficos* contiene las clases siguientes:

- *RunPanel*: extiende el *JPanel* de Java y añade un bucle de repintado al panel que hace de lienzo.

- *VisualPanel*: extiende *RunPanel* y es el centro neurálgico del desarrollo de nuestro apartado gráfico. Incluye los atributos necesarios para almacenar las posiciones de todos los nodos de un árbol, así como todas las secciones en que deba dividirse una estructura para poder ser animada de forma correcta.

- *Animación*: clase que facilita la identificación de cada tipo de animación. Incluye un nombre, los parámetros necesarios para realizarse y el árbol sobre el que se va a realizar la animación.

3.2.1.1.3. Interacción con entorno gráfico

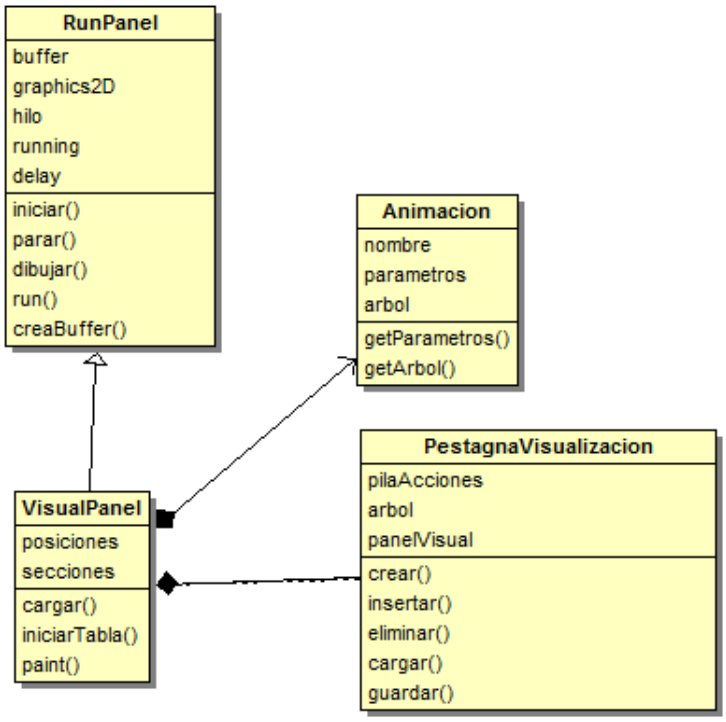

Figura 6. Diagrama de interacción con el entorno gráfico

La conexión entre los algoritmos de ambas estructuras y las clases que implementan la funcionalidad gráfica de la herramienta se lleva a cabo mediante la clase PestagnaVisualización. Esta clase incluye un objeto de tipo VisualPanel, descrito en el apartado anterior y permite incluir la funcionalidad gráfica a las pizarras.

3.2.1.1.4. Tests

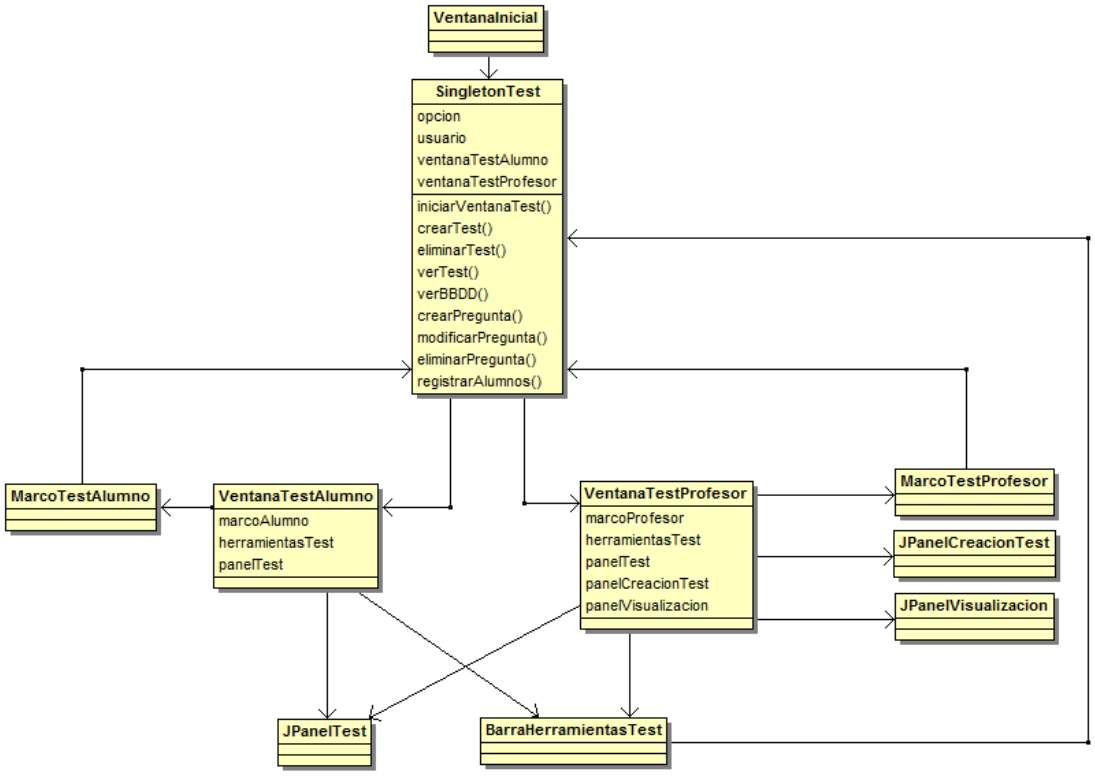

Figura 7. Diagrama del paquete Test

El paquete *test* incluye un amplio conjunto de clases que facilitan la separación de los distintos modos de ejecución de la herramienta. La clase *SingletonTest* será la clase encargada de recoger la opción escogida por el usuario y de activar las ventanas de alumno o profesor según convenga. Además, esta clase posee los métodos necesarios para crear, eliminar y visualizar test de la base de datos así como crear y modificar preguntas y, por último, registrar alumnos.

La clase *VentanaTestAlumno* restringe el uso de la herramienta únicamente a la resolución de test y al uso de la pizarra. *VentanaTestProfesor* activa todas las operaciones ya mencionadas de la herramienta.

3.2.1.1.5. Interfaz

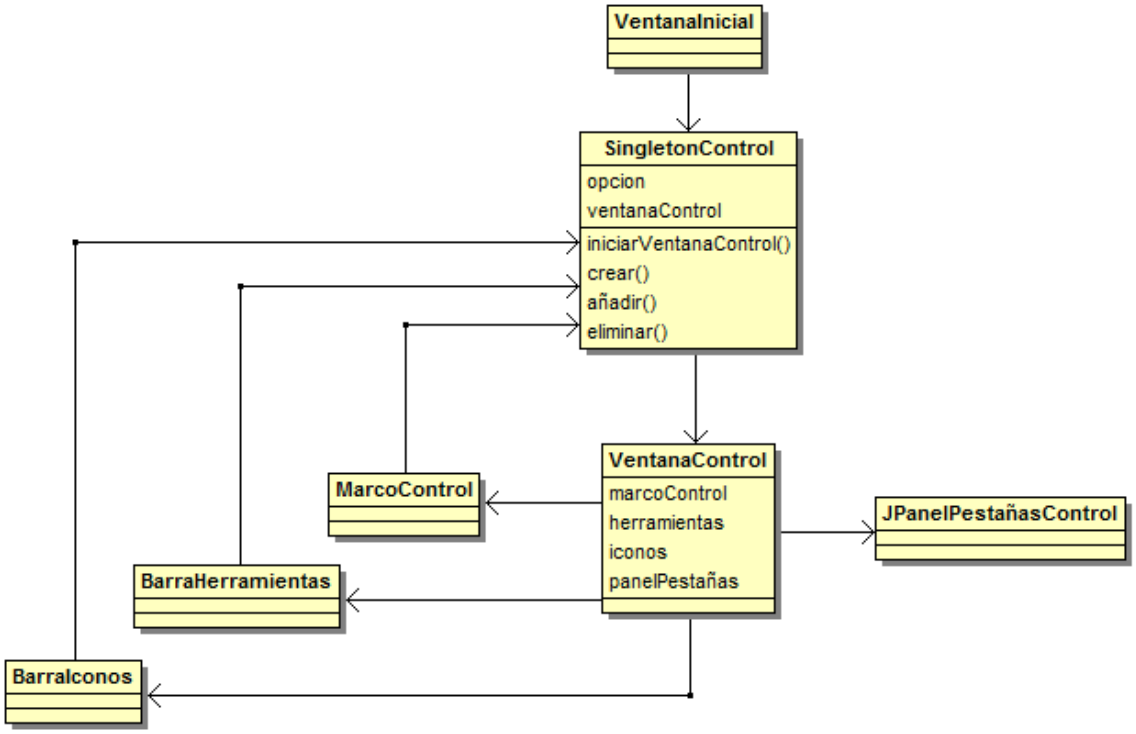

Figura 8. Diagrama de la interfaz

La clase que controla todo el entorno de interfaces es la clase *SingletonControl*. Al igual que hacia *SingletonTest*, esta clase almacena las opciones escogidas por el usuario y muestra las interfaces correspondientes en función de la elección. Incluye además las llamadas a las funciones de inserción y eliminación de los árboles. Podemos decir que las clases *Singleton* controlan el flujo de información de toda la herramienta.

3.2.1.2. Casos de uso

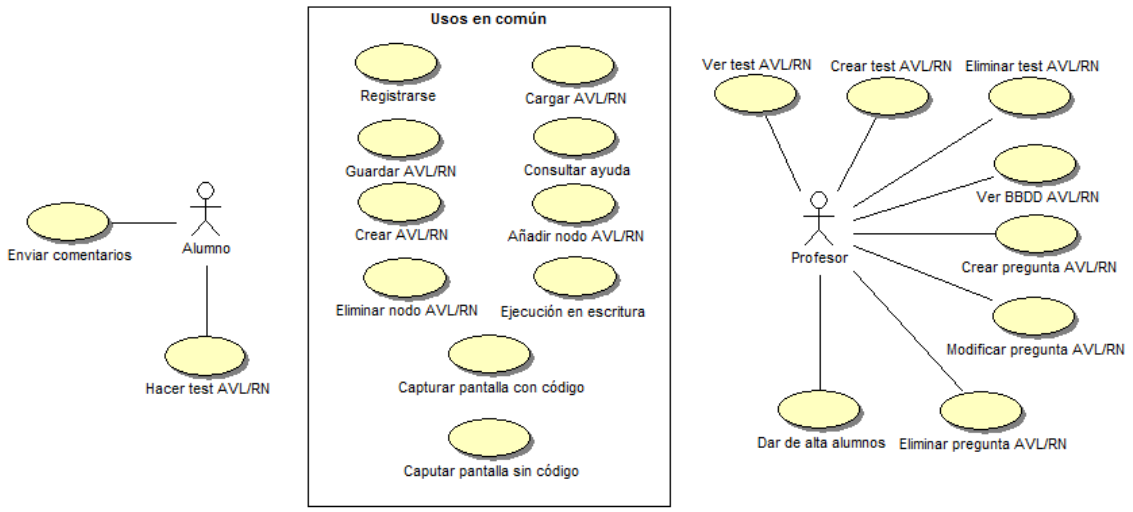

Figura 9. Diagrama de los Casos de uso

Según quien sea el tipo de usuario que controla la herramienta, éste podrá realizar distintas acciones. Si el usuario es un alumno, este podrá realizar las acciones recogidas en la columna izquierda de la imagen anterior más todas las que aparecen en la columna central, es decir, el uso de la herramienta queda restringido en cuanto a modificación de las bases de datos de preguntas y alumnos se refiere; por lo demás, es plenamente funcional.

- Crear AVL/Rojinegro.
- Guardar/Cargar AVL/Rojinegro.
- Insertar/Eliminar AVL/Rojinegro.
- Secuencias de instrucciones AVL/Rojinegro.
- Resolver tests.
- Escribir comentarios al profesor.
- Capturar imágenes.

Si el usuario es un profesor, todo lo que quedaba restringido al alumno pasa a formar parte de la labor de este. Podrá realizar todo tipo de acciones sobre las bases de datos de preguntas, tests y/o alumnos, así como utilizar la parte común con los alumnos, es decir, la pizarra gráfica en sus modos distintos.

- Insertar/Eliminar/Modificar preguntas de la base de datos.
- Crear/Eliminar/Modificar tests de la base de datos.
- Registrar/Eliminar alumnos de la base de nombres.
- Crear AVL/Rojinegro.
- Guardar/Cargar AVL/Rojinegro.
- Insertar/Eliminar AVL/Rojinegro.
- Capturar imágenes.
- Secuencias de instrucciones AVL/Rojinegro.

3.2.2. Implementación

En esta sección, vamos a explicar cómo se han implementado cada uno de los apartados de la herramienta **V-Tree**. Iremos desde las estructuras de datos que ofrece hasta las interfaces que hacen posible su manejo, pasando por las novedosas animaciones y el entorno de desarrollo de tests.

3.2.2.1. Lenguaje y entorno de desarrollo

Por un lado, el lenguaje de programación escogido para la elaboración de **V-Tree** ha sido Java. Esta decisión se tomó, además de por su simplicidad y amplias alternativas, con la idea de reutilizar parte del código fuente procedente de los proyectos Vedya anteriores, aunque finalmente esta reutilización no ha sido posible.

Por otra parte, el entorno de programación que se ha utilizado para el desarrollo de la aplicación ha sido *Eclipse Platform*, tanto para la parte algorítmica y gráfica, como para el propio diseño de las interfaces. En concreto, la versión utilizada es *Eclipse Helios.*

3.2.2.2. Árboles binarios de búsqueda

Comenzamos entonces hablando sobre los dos tipos de estructuras de datos arborescentes que ofrece **V-Tree**. La implementación de ambas difiere en pocos puntos ya que ambas implementaciones se basan en añadir restricciones a un árbol binario de búsqueda.

Un árbol binario de búsqueda es un árbol binario (cada nodo sólo tiene dos hijos) en el que cada nodo del subárbol izquierdo es menor que la raíz del subárbol y en el que cada nodo del subárbol derecho es mayor que la raíz del subárbol. Los algoritmos de inserción y eliminación en este tipo de estructuras se apoyan en el algoritmo de búsqueda binaria. La decisión de escoger la siguiente rama a explorar se toma comparando el elemento que queremos encontrar con el nodo actual, si éste es mayor, se continua explorando la rama derecha, en caso contrario, se explora la rama izquierda. El proceso se repite hasta encontrar el dato pretendido o hasta llegar a una hoja.

Supongamos que queremos buscar el nodo 5 en el siguiente árbol. Veamos el proceso de búsqueda.

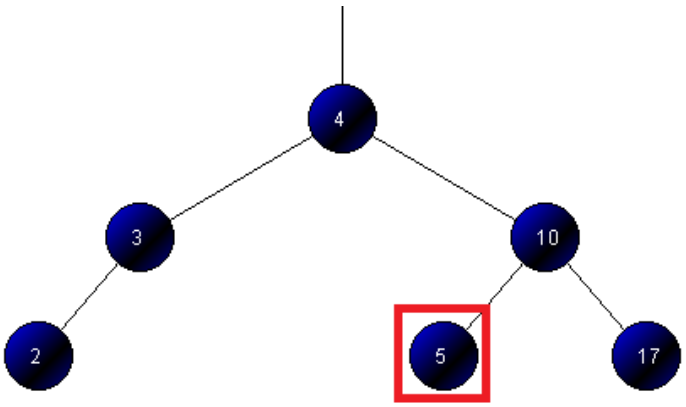

Figura 10. Árbol sobre el que buscar el nodo 5

Al comparar el nodo 4 con el 5, continuará explorando el hijo derecho y tras comparar con el 10, decidirá explorar el hijo izquierdo. Tras esto, encontrará el nodo buscado.

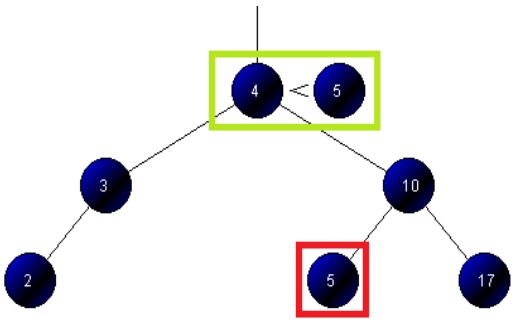

Figura 11. Comparación con el nodo Raíz

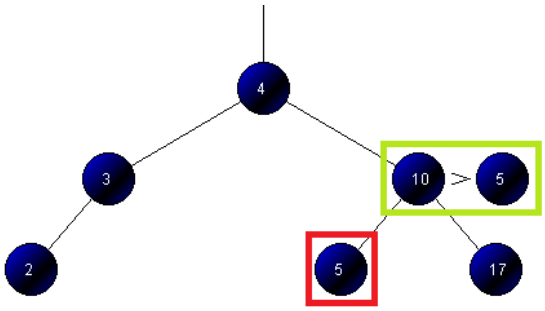

Figura 12. Comparación con el hijo derecho

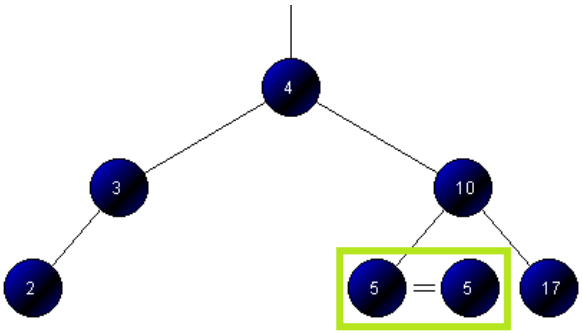

Figura 13. Nodo encontrado

3.2.2.3. Módulo AVL

Un árbol AVL es un árbol binario de búsqueda que cumple la propiedad de que la diferencia de alturas de los subárboles de cada uno de los nodos es, a lo sumo, uno. A esta propiedad se le denomina habitualmente “estar equilibrado”.

El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, Adelson-Velskii y Landis. Lo dieron a conocer en la publicación de un artículo en 1962: *"An algorithm for the organization of information"*.

En cuanto a la eficiencia de las operaciones sobre este tipo de estructuras, se asegura que el tiempo de ejecución de las operaciones sobre árboles AVL es, en el caso peor, logarítmico respecto al número de nodos.

La propiedad de "estar equilibrado" asegura que la profundidad del árbol sea también logarítmica respecto al número de nodos.

A cambio de mantener estas ventajas, las operaciones de inserción y eliminación se vuelven más complejas. Encontrar el lugar en el que insertar, o el nodo que se debe eliminar, es muy sencillo, pero puede provocar la pérdida de la propiedad "estar equilibrado", por lo que es necesario tratar este problema en el mismo momento en que se produce.

Pasamos entonces a mostrar cómo hemos representado un árbol AVL y cómo se han tratado cada uno de los nuevos problemas a resolver a nivel de implementación.

De entre las distintas posibles estrategias para representar esta estructura de datos, nos decidimos por la idea de que un AVL sería un nodo que dispondría del valor del dato que queremos almacenar, de un hijo izquierdo y un hijo derecho que serán también a su vez árboles AVL (por lo que contendrán a su vez esta misma información) y, por último, del valor de la altura de cada nodo del árbol. Este último dato será la clave para decidir en qué momentos se producen rotaciones y de qué tipo. Los atributos de la clase son entonces los siguientes:

```
private Comparable elemento;
private AVL hijoIzdo;
private AVL hijoDerecho;
private int altura;
```

Los algoritmos más importantes que se han implementado son *Insertar* y *Eliminar*. Son algoritmos recursivos que recorren el árbol en busca del lugar donde se debe colocar el nuevo elemento, o por el contrario, eliminarlo. Ambos comprueban si el árbol resultante se ha desequilibrado, para ello, comparan la diferencia de alturas entre los hijos derecho e izquierdo y, si esta diferencia es de dos unidades, proceden a su reequilibrio, usando para ello los algoritmos de rotación clásicos [3]. Además, actualizarán las alturas en cada uno de los nodos en caso de que sea necesario.

Cabe destacar que la diferencia de alturas entre dos nodos hermanos no puede ser superior a dos unidades, debido a que los elementos del árbol se insertan y eliminan uno a uno, no pudiéndose producir casos más complejos de desequilibrios.

```
private AVL inserta(AVL avl, Comparable elem){

        if ( avl == null) avl = new AVL(elem);

        else if (compara(avl.elemento,elem)<0){

                avl.hijoIzdo = inserta(avl.hijoIzdo,elem);
               if(profundidad(avl.hijoIzdo) - profundidad(avl.hijoDerecho) == 2)
                       if (compara(avl.hijoIzdo.elemento,elem,pasos)<0)avl = rotarIzda(avl);
                       else    avl = rotarDobleIzda(avl);

        }else if(compara(avl.elemento,elem) > 0){

                avl.hijoDerecho = inserta(avl.hijoDerecho,elem);
                if (profundidad(avl.hijoDerecho)-profundidad(avl.hijoIzdo) == 2)
               if (compara(avl.hijoDerecho.elemento,elem,pasos)>0)avl = rotarDrcha(avl);
               else    avl = rotarDobleDrcha(avl);

        }else{}//elemento duplicado

        avl.altura = max(profundidad(avl.hijoIzdo),profundidad(avl.hijoDerecho))+1;
        return avl;
}
```

En el algoritmo de eliminación, hay que tener en cuenta que al eliminar un nodo, otro tiene que ocupar su lugar. Esa decisión la tomarán los métodos *subeHijoIzdo* y *subeHijoDrcho.* Estos métodos calculan el máximo de los nodos del hijo izquierdo y el mínimo de los nodos del hijo derecho de un árbol, respectivamente.

```
public AVL elimina(AVL avl,Comparable elem){

    if ( avl == null) return avl;
   else if (compara(avl.elemento,elem) < 0)avl.hijoIzdo=elimina(avl.hijoIzdo,elem);
   else if (compara(avl.elemento,elem) > 0)avl.hijoDerecho=elimina(avl.hijoDerecho,elem);
   else{
      if( esHoja(avl))return null;
      else if (profundidad(avl.hijoDerecho) < profundidad(avl.hijoIzdo))
           avl = subeHijoIzdo(avl);
      else    avl = subeHijoDrcho(avl);
   }
   avl.altura=max(profundidad(avl.hijoIzdo),profundidad(avl.hijoDerecho))+1;
   if(profundidad(avl.hijoIzdo) - profundidad(avl.hijoDerecho) == 2)
      if (profundidad(avl.hijoIzdo.hijoDerecho)<=profundidad(avl.hijoIzdo.hijoIzdo)){
                           avl = rotarIzda(avl);
      }
      else avl = rotarDobleIzda(avl);
   else if (profundidad(avl.hijoDerecho) - profundidad(avl.hijoIzdo) == 2)
      if(profundidad(avl.hijoDerecho.hijoDerecho)>=profundidad(avl.hijoDerecho.hijoIzdo)){
              avl = rotarDrcha(avl);
      }
      else    avl = rotarDobleDrcha(avl);

   return avl;
}
```

```
private AVL subeHijoIzdo(AVL avl) {

     if(avl.hijoDerecho == null)avl = avl.hijoIzdo;
    else if(avl.hijoIzdo.hijoDerecho == null){
            avl.hijoIzdo.hijoDerecho=avl.hijoDerecho;
            avl = avl.hijoIzdo;
    }
    else{
            AVL avlAux = maxAVL(avl.hijoIzdo);
            AVL enlace = new AVL<Comparable>(avlAux.elemento);
            enlace.hijoIzdo = avl.hijoIzdo;
            enlace.hijoDerecho = avl.hijoDerecho;
            avl = enlace;
            avl.hijoIzdo = elimina(avl.hijoIzdo,avlAux.elemento,pasos);

    }
    return avl;
}
```

```
private AVL subeHijoDrcho(AVL avl) {

   if(avl.hijoIzdo == null)avl = avl.hijoDerecho;
    else if(avl.hijoDerecho.hijoIzdo == null){
            avl.hijoDerecho.hijoIzdo=avl.hijoIzdo;
            avl = avl.hijoDerecho;
    }
    else{
            AVL avlAux = minAVL(avl.hijoDerecho);
            AVL enlace = new AVL<Comparable>(avlAux.elemento,pasos);
            enlace.hijoIzdo = avl.hijoIzdo;
            enlace.hijoDerecho = avl.hijoDerecho;
            avl = enlace;
            avl.hijoDerecho = elimina2(avl.hijoDerecho,avlAux.elemento,pasos);

    }
    return avl;
}
```

Una vez descritos los métodos de *inserción y eliminación*, es momento de explicar cada una de las rotaciones que podemos necesitar para devolver a un AVL su propiedad de "estar equilibrado". Tenemos cuatro casos de desequilibrio, que explicaremos tomando un ejemplo concreto y provocando cada situación por separado:

- Rotación izquierda (**rotación LL**): al insertar el elemento 10 en el árbol, se produce una diferencia de altura de dos unidades en el hijo izquierdo. El nodo 15 pasa a ser la raíz, el nodo 20 pasa a ser el hijo derecho de 15 y el 10 se queda como hijo izquierdo de 15.

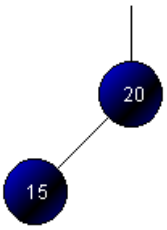

Figura 14. Árbol inicial al que insertar el elemento 10

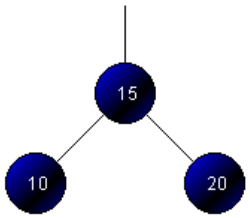

Figura 15. Árbol resultante después de una rotación LL

```
private AVL rotarIzda(AVL avl){

        AVL a1 = avl.hijoIzdo;
        avl.hijoIzdo = a1.hijoDerecho;
        a1.hijoDerecho = avl;
        avl.altura = max( profundidad(avl.hijoIzdo),profundidad(avl.hijoDerecho)) + 1;
        a1.altura= max( profundidad( a1.hijoIzdo ), avl.altura ) + 1;
        return a1;
}
```

- Rotación derecha (**rotación RR**): al insertar el elemento 20 en el árbol, se produce una diferencia de altura de dos unidades en el hijo derecho. En este caso, el 15 pasa a ser el nodo raíz, el nodo 10 el hijo izquierdo de 15 y el nodo 20 mantiene su posición como hijo derecho de 15.

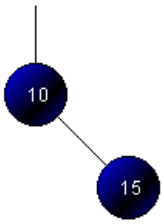

Figura 16. Árbol inicial antes de insertar el elemento 20

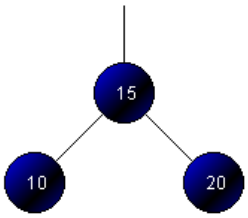

Figura 17. Árbol resultante después de una rotación RR

```
private  AVL rotarDrcha(AVL avl){

       AVL a1 = avl.hijoDerecho;
      avl.hijoDerecho = a1.hijoIzdo;
      a1.hijoIzdo = avl;
      avl.altura = max(profundidad(avl.hijoIzdo), profundidad(avl.hijoDerecho)) + 1;
      a1.altura = max( profundidad( a1.hijoDerecho ), avl.altura ) + 1;
      return a1;

 }
```

- Rotación doble derecha (**rotación RL**): al insertar el elemento 15 en el árbol, se produce un desequilibrio que no puede arreglarse con una rotación simple porque el nodo 15 se coloca como hijo izquierdo del nodo 20. Se procede primero a rotar a la izquierda desde el nodo 20 y luego a la derecha desde el nodo 10.

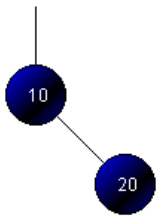

Figura 18. Árbol inicial antes de insertar el elemento 15

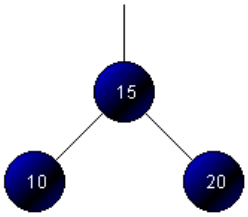

Figura 19. Árbol resultante después de una rotación RL

```
private AVL rotarDobleDrcha(AVL avl){

      AVL derecho = rotarIzda(avl.hijoDerecho);
      avl.hijoDerecho = derecho;
      avl = rotarDrcha(avl);
      return avl;
}
```

- Rotación doble izquierda (**rotación LR**): al insertar el elemento 8 en el árbol, se produce un desequilibrio que tampoco puede arreglarse con rotaciones simples ya que éste se coloca como hijo derecho del nodo 5. En este caso se procede primer a rotar hacia la derecha desde el nodo 5 y luego a la izquierda desde el nodo 10.

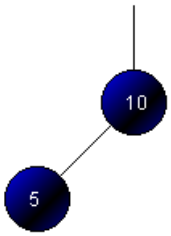

Figura 21. Árbol inicial antes de insertar el elemento 8

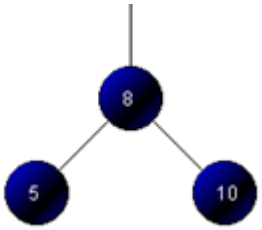

Figura 20. Árbol resultante después de una rotación LR

```
private AVL rotarDobleIzda(AVL avl){

      AVL izquierdo = rotarDrcha(avl.hijoIzdo);
      avl.hijoIzdo = izquierdo;
      avl = rotarIzda(avl);
      return avl;

}
```

Una vez explicados los algoritmos y sus aplicaciones sobre casos sencillos, vamos a reproducir alguno de los casos anteriores sobre un árbol AVL de mayores dimensiones. Comenzamos con el siguiente árbol equilibrado, que llamaremos "árbol base" de aquí en adelante.

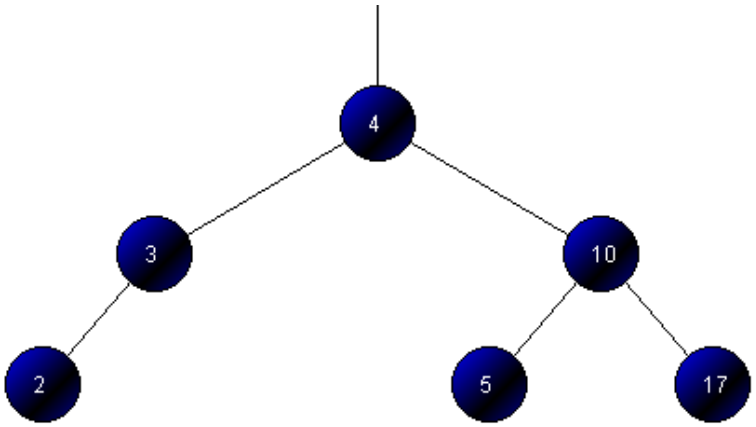

Figura 22. Árbol base

Probemos a insertar un nodo con valor 1.

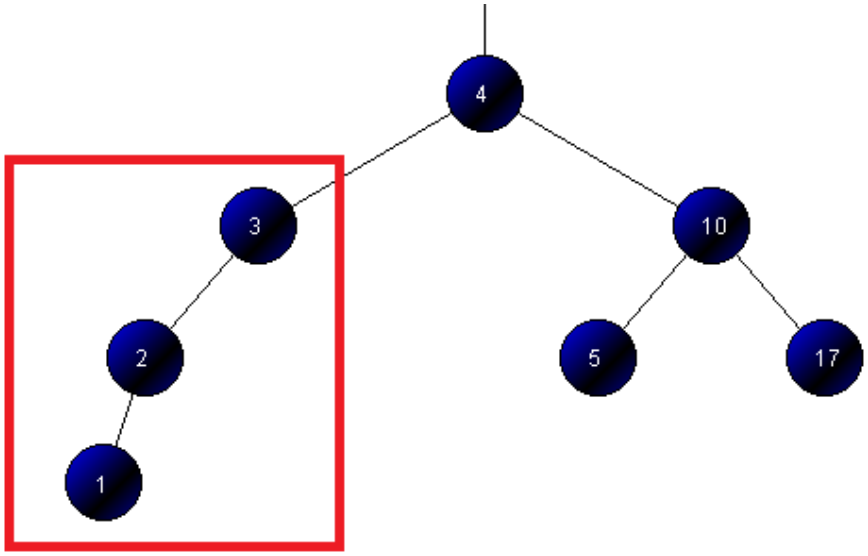

Figura 23. Árbol base con el nodo de valor 1 insertado y desequilibrado

Podemos observar que se produce un desequilibrio en el nodo 3, puesto que la profundidad de su hijo izquierdo es dos unidades mayor que la del hijo derecho. Se procede pues a equilibrar el subárbol aplicando una rotación de tipo LL. El árbol resultante es:

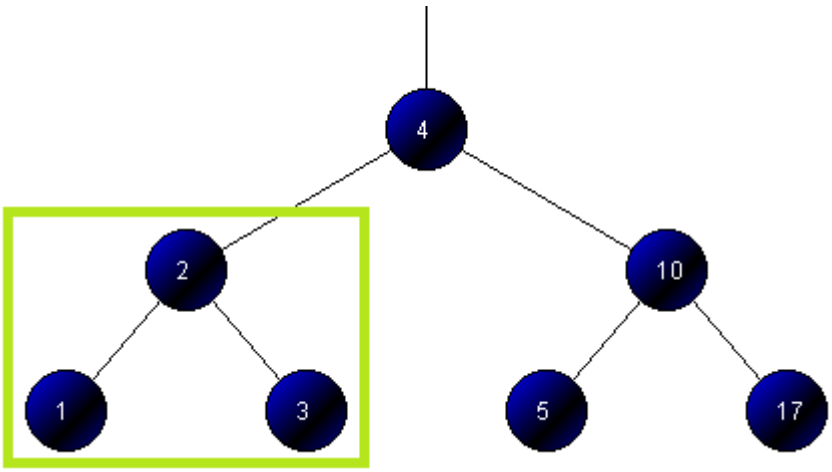

Figura 24. Árbol resultante tras una rotación LL para equilibrarlo

Tomando de nuevo el árbol base, eliminemos el nodo 4. Se trata de la raíz del árbol, el algoritmo de eliminación debe escoger el nodo que ocupará la posición de la raíz, lo decidirá con el método *subeHijoDerecho* explicado anteriormente.

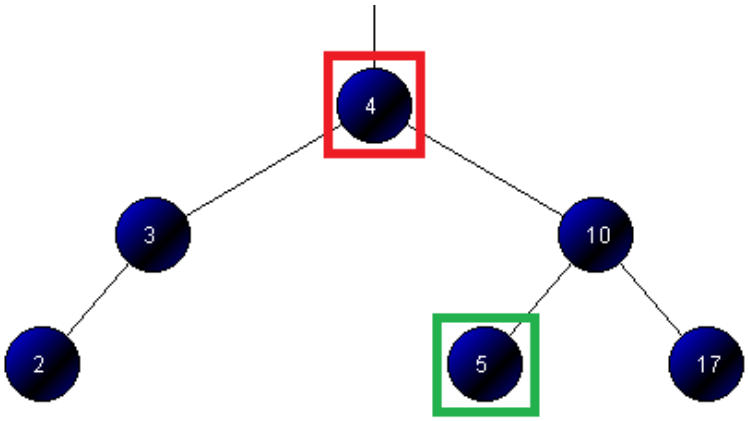

Figura 25. Árbol base al que eliminar el nodo 4

En este caso, la nueva raíz será el nodo 5, ya que es el mínimo de los valores del hijo derecho del nodo a eliminar. El árbol resultante está equilibrado y no es necesario realizar rotación alguna.

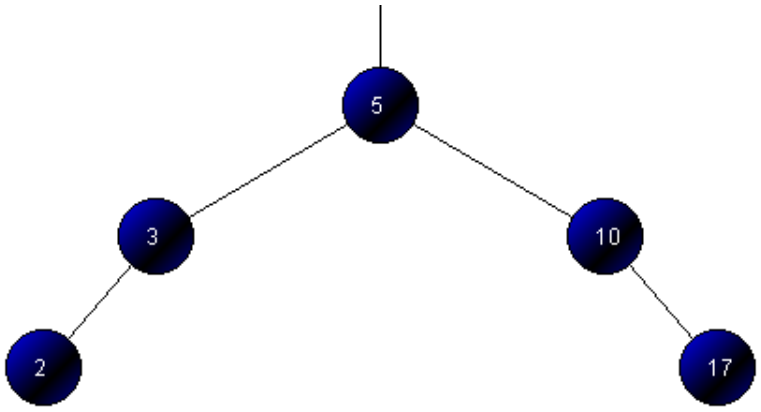

Figura 26. Árbol resultante tras eliminar el nodo 4 y sustituirlo por su sucesor

Como último ejemplo, veremos que si insertamos el nodo 12, produciremos una diferencia de alturas entre los hijos del nodo 10. Este desequilibrio no podrá arreglarse con una rotación simple.

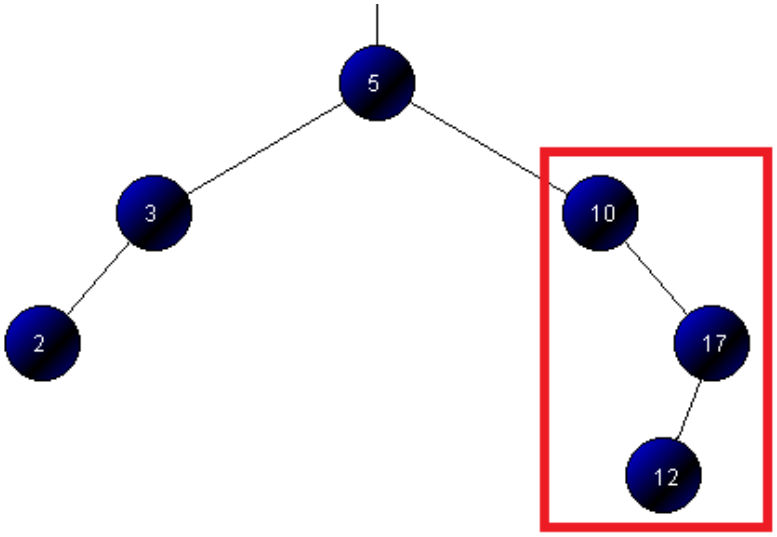

Figura 27. Árbol base que necesita una rotación doble LR para equilibrarlo

Aplicaremos una rotación doble izquierda (LR) para devolver al árbol la condición de equilibrio.

1. Rotación izquierda (LL) : aplicamos el algoritmo de rotación LL para obtener el siguiente árbol intermedio, el nodo 10 sigue desequilibrado.

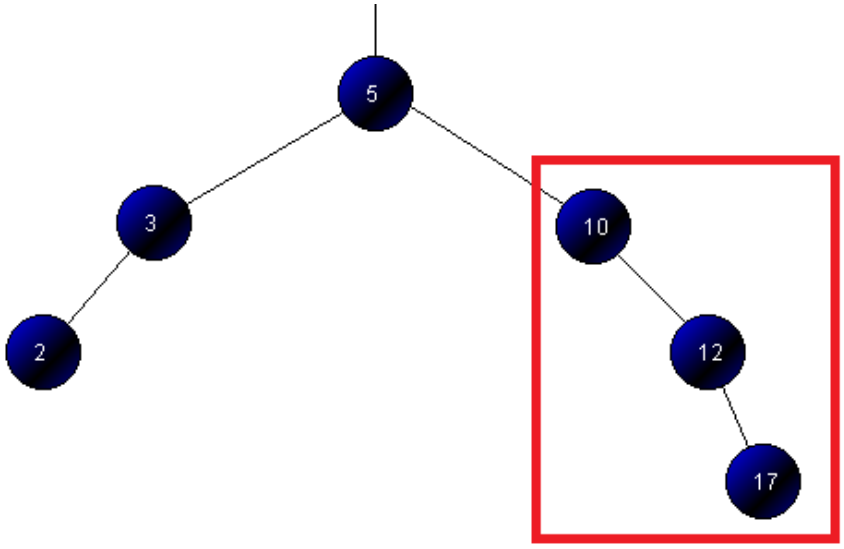

Figura 28. Árbol resultante tras la primera rotación LL

2. Rotación derecha (RR):

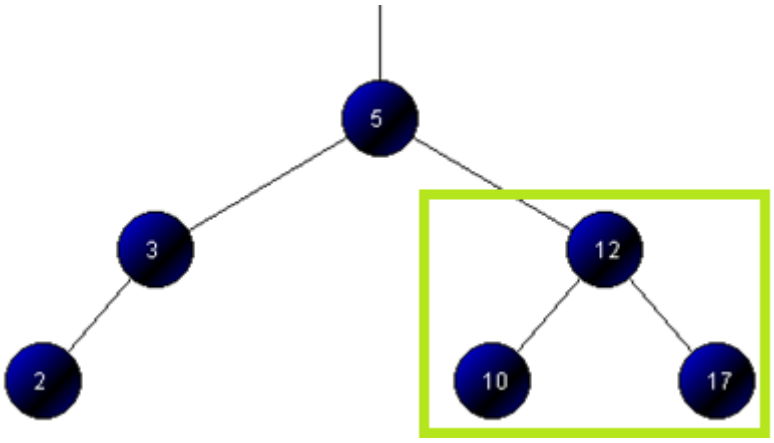

Figura 29. Árbol equilibrado resultante tras la segunda rotación RR

3.2.2.4. Módulo Rojinegro

Un árbol rojo-negro es un árbol binario de búsqueda que posee además, para cada nodo del árbol, un atributo *"color",* que se utiliza para añadir las siguientes restricciones a un árbol de este tipo:

1. Todos los nodos son rojos o negros.
2. Los hijos nulos se consideran nodos negros.
3. La raíz es siempre un nodo negro.
4. Todo nodo rojo tiene padre negro.
5. Todos los caminos desde la raíz hasta las hojas tiene el mismo número de nodos negros.
6. El recorrido que pasa por el número máximo de nodos, nunca puede pasar por más del doble de nodos que se recorren por el recorrido que pasa por el mínimo número de nodos.

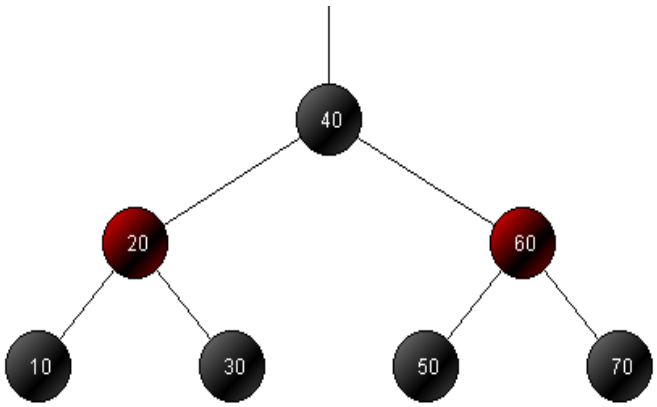

Figura 30. Ejemplo de árbol Rojo-Negro

La peculiaridad de este tipo de árboles reside además en que, a la hora de insertar o eliminar un elemento, la condición de equilibrio no se recupera con una rotación tal y como se hacía en los árboles AVL si no que ahora puede ser necesario cambiar el color de un conjunto de nodos para recuperar la situación de equilibrio inicial.

La forma de representar esta estructura es la misma que para los árboles AVL, salvo que añadimos el atributo "*color"*, necesario para poder comprobar las restricciones descritas y realizar las rotaciones y cambios de color pertinentes.

Sólo nos queda pues, comentar los cambios introducidos en los algoritmos de inserción y eliminación así como la implementación del método *cambioColor*. Haremos especial hincapié en este último algoritmo y en la forma de detectar cuándo debemos aplicarlo.

Cuando se inserta un nodo en el árbol rojo-negro, lo primero que hay que tener en cuenta es que el nodo que se inserta siempre tiene que ser rojo. Partiendo de este hecho y suponiendo que ese nodo ya está insertado veremos una serie de comprobaciones y rotaciones que llevan a cabo el proceso de equilibrado del árbol.

El caso mas obvio se da cuando el padre del elemento insertado no existe. En ese caso se puede dar por hecho que estamos en la raíz, por lo que se cambia el color del nodo insertado a negro para no violar la segunda ley y se finaliza la inserción. Comencemos creando la raíz de nuestro árbol rojo-negro.

Figura 31. Árbol Rojo-Negro tras la inserción del primer nodo

Si el padre existe, lo siguiente que se comprueba es si el padre del nodo insertado es negro. En caso de ser así, ya hemos acabado. Para el par de nodos nulos (negros) nuevos que se han formado por el elemento insertado, cuando recorremos desde la raíz hasta cada uno de éstos nodos, se puede comprobar fácilmente que el número de nodos es exactamente el mismo. Como el nodo insertado es rojo, no aumenta el número de nodos negros en esa rama, y como el padre es negro, no se viola ninguna de las reglas que hacen que un árbol sea del tipo rojo-negro. Este caso podemos recrearlo insertando un par de nodos con los valores 25 y 75 en el árbol anterior y, como vemos, no violamos ninguna de las restricciones impuestas por lo que no es necesario realizar equilibrado alguno.

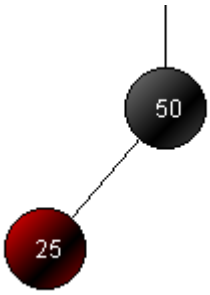

Figura 32. Árbol resultante tras insertar el nodo 25

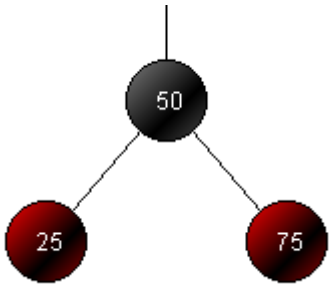

Figura 33. Árbol resultante tras insertar el nodo 75

Si nos plantemos insertar ahora el nodo 15, provocaremos la violación de la regla 4: todo nodo rojo tiene padre negro.

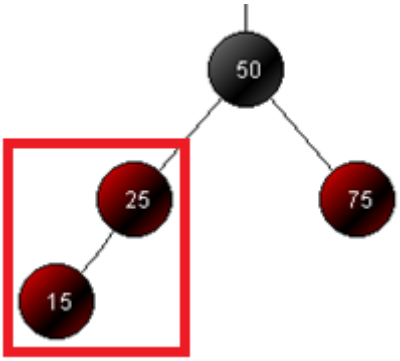

Figura 34. Árbol desequilibrado tras insertar el nodo 15

Debemos aplicar un cambio de color en el nodo 25, pero esto provocaría la violación de la regla 5: Todos los caminos desde la raíz hasta las hojas tiene el mismo número de nodos negros. Miramos entonces el hermano del nodo en el que se ha producido la inserción, como es de color rojo, también debemos cambiarlo a color negro, de esta forma, desde la raíz a las hojas tenemos el mismo número de nodos negros. ¿Por qué no cambió de color el nodo 50? Porque se incumpliría la primera regla, la raíz es un nodo negro.

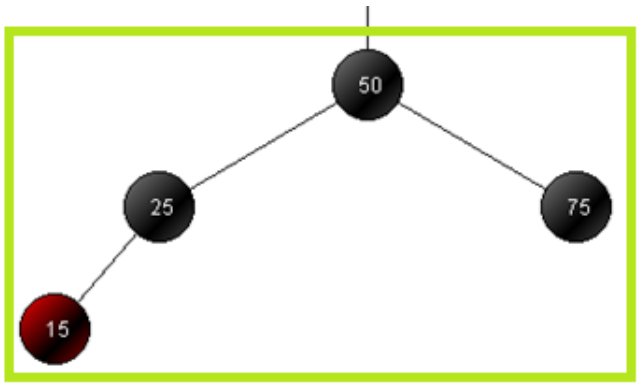

Figura 35. Árbol equilibrado tras un cambio de color

Vamos a provocar una rotación simple con un cambio de color, para ello introducimos un nodo de valor 7, por ejemplo, para que se sitúe como hijo izquierdo del elemento 15.

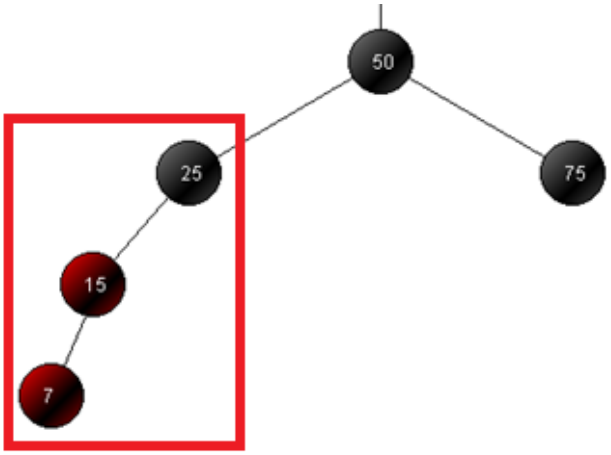

Figura 36. Árbol desequilibrado tras insertar el nodo 7

El hermano del nodo en que se realiza la inserción no existe y, por tanto, se considera negro. Como la inserción se realiza en el hijo izquierdo del subárbol izquierdo se aplica una rotación simple con cambio de color.

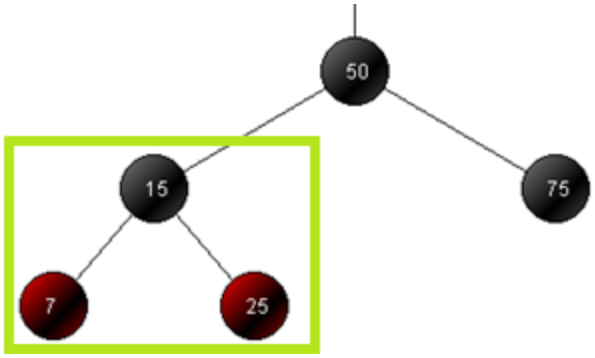

Figura 37. Árbol equilibrado tras una rotación y un cambio de color

A continuación, provocaremos otro desequilibrio incluyendo el elemento 11.

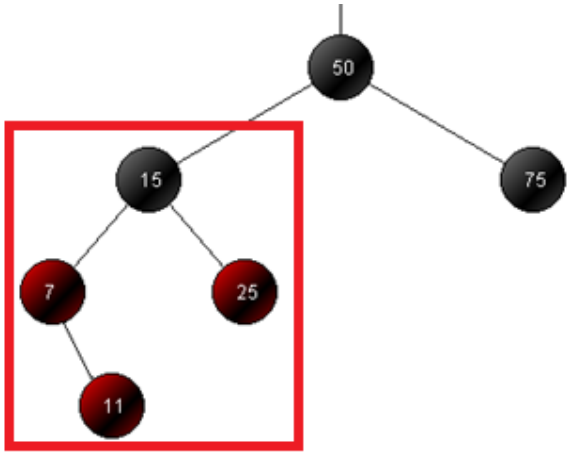

Figura 38. Árbol desequilibrado tras la inserción del nodo 11

Como el hermano del nodo en el que se realiza la inserción es rojo, aplicamos un cambio de color, de tal forma que 15 pase a ser rojo, 7 y 25 negros y 11 se queda rojo.

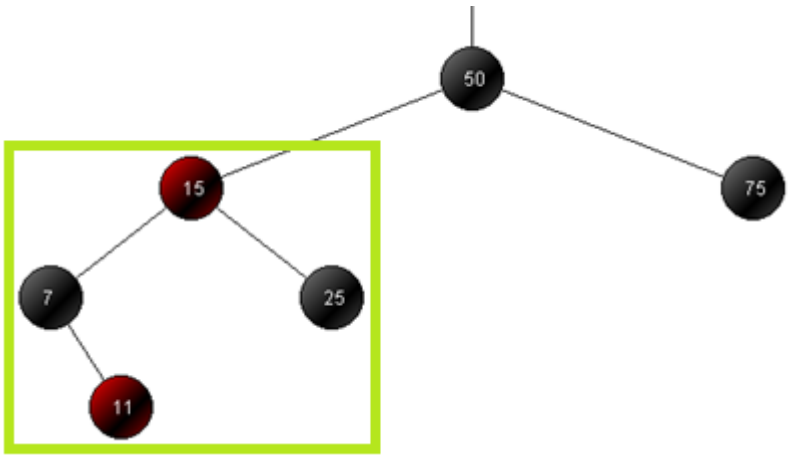

Figura 39. Árbol equilibrado tras tres cambios de color

Al insertar el elemento 60, el árbol mantiene todas las condiciones de equilibrio, no es necesario aplicar ninguna rotación ni ningún cambio de color. Si a continuación añadimos el 65 tendremos la siguiente situación de desequilibrio:

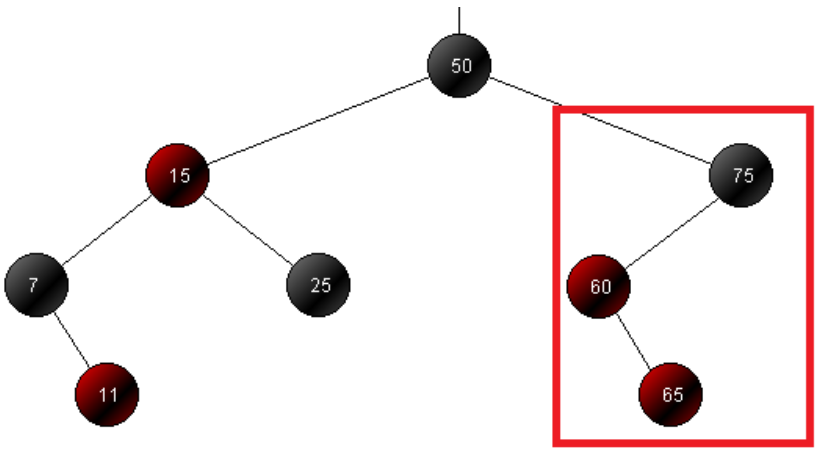

Figura 40. Árbol desequilibrado tras la inserción del nodo 60 y el nodo 65

Tendrá que solucionarse mediante una rotación doble y un cambio de color.

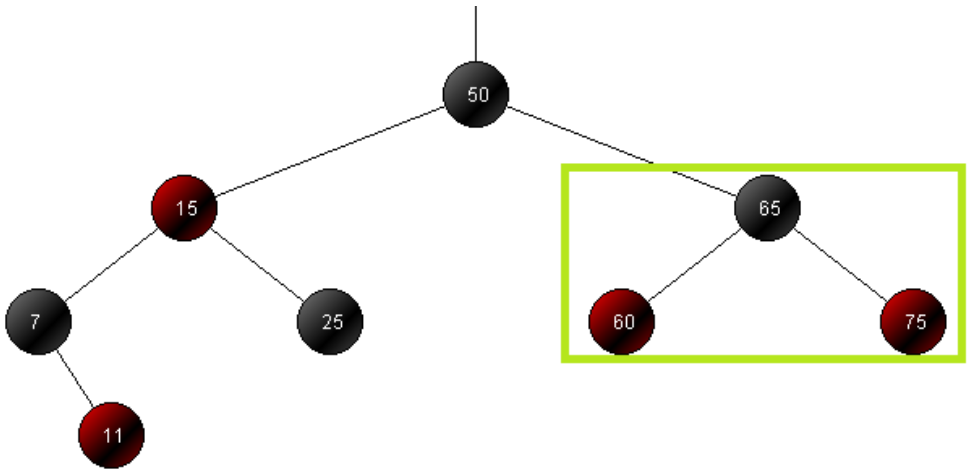

Figura 41. Árbol equilibrado tras dos rotaciones y un cambio de color

Con este último caso, hemos conseguido ilustrar los diferentes tipos de proceso de equilibrado de un árbol rojo-negro. A continuación se adjunta el código Java que implementa este algoritmo. Por claridad, se han eliminado las instrucciones que conciernen a la creación de la secuencia de acciones gráficas asociadas a este tipo de árbol.

```
private RN inserta(RN rn, Comparable elem) {

     if (rn == null) {
        rn = new RN(elem);
        rn.NEGRO = false;
     } else if (compara(rn.elemento, elem) < 0) {
        rn.hijoIzdo = inserta2(rn.hijoIzdo, elem);
        if(!equilibrio)
            equilibrio=equilibraInsercion(rn.hijoIzdo);
        } else if (compara(rn.elemento, elem) > 0) {

            rn.hijoDerecho = inserta(rn.hijoDerecho, elem);

            if(!equilibrio)
               equilibrio=equilibraInsercion(rn.hijoDerecho);
            }
            rn.altura = max(profundidad(rn.hijoIzdo), profundidad(rn.hijoDerecho)) + 1;
           if(nodoSustituido==rn)
              return nodoSustituto;
           else
              return rn;
       }
```

Por último, explicaremos el proceso de equilibrado en este tipo de árboles cuando eliminamos elementos del árbol.

Para explicar el proceso de equilibrado de la eliminación, vamos a dar por hecho que el algoritmo de búsqueda ya ha localizado el nodo a borrar. En el caso de que sea un nodo con hijos, localizaremos al nodo sucesor (nodo mínimo del hijo derecho) o antecesor (nodo máximo del hijo izquierdo), dependiendo de la rama mas profunda, y sustituiremos el valor del nodo actual por el valor del nodo encontrado (dejando el color que tenía el nodo original eliminado), después volveremos a llamar al proceso eliminador para que elimine el nodo que ha sustituido. Como no hemos modificado la estructura del árbol ni hemos cambiado ninguno

de sus colores, ninguna de las propiedades de los árboles rojo-negros se ha visto afectada, por lo que no hay que hacer ningún tipo de proceso de equilibrado.

Ahora supongamos el caso en el que nos encontramos encima de un nodo que se debe eliminar y no sustituir. En estos casos podemos siempre dar por hecho que el nodo a eliminar tiene a lo sumo un hijo, de no ser así se debería sustituir el nodo por un sucesor o por un antecesor.

El caso más trivial se da cuando el nodo que se borra es rojo. En este caso sustituimos el nodo por su hijo o lo eliminamos si no tiene ningún hijo. Como los nodos rojos no afectan al equilibrio del árbol, no es necesario ningún reajuste a la hora de eliminarlos.

Cuando el nodo a eliminar es negro equilibraremos el árbol en función del nodo que ha sustituido al nodo eliminado. Para el caso en el que el nodo borrado sea una hoja (un nodo sin hijos), hemos de recordar que los nodos nulos se consideran negros. En este caso primero debemos ver el color del nodo, si es rojo, se cambia a negro y hemos conseguido mantener el número de nodos negros que había.

En caso de ser negro también hallamos al padre del nodo que ha sustituido al nodo antiguo. Si éste no existe, significa que el nodo que hemos eliminado es el nodo raíz. En este caso, el nodo que ha sustituido al nodo raíz es nulo o rojo. Si no fuera nulo o rojo significaría que el nodo, antes de ser eliminado, tendría dos hijos, y esto provocaría una sustitución por un nodo sucesor o antecesor. Si fuese rojo, lo habríamos cambiado negro en el paso anterior y no habría ningún problema, en caso de ser nulo, el árbol resultante es un árbol vacío y tampoco habría que corregir nada.

Si existe el padre, hallamos al nodo hermano y comprobamos el color de estos dos. En caso de que el padre sea negro y el hermano rojo, realizamos una rotación posicionando al hermano como padre, e intercambiamos el color de estos dos para que no haya ninguna incoherencia en cuanto a todos los nodos hijo por debajo de estos. Como al realizar una rotación, el hijo del hermano mas próximo al nodo eliminado pasa a ser el hijo del padre y hermano del nodo sustituto, nos aseguramos que este nuevo nodo hermano sea negro (puesto que el anterior hermano era rojo y sus hijos tienen que ser negros por definición).

A partir de este paso tenemos que tener en cuenta el color de los nodos primos (hijos del nodo hermano).

Si el padre es negro, el hermano es negro y ambos primos también son negros, cambiamos el color del hermano a rojo (para igualar el número de nodos negros) y volvemos a equilibrar el árbol desde el principio a partir del nodo padre, ya que ahora toda esa rama tiene un nodo negro menos (el que hemos borrado por un lado y el que acabamos de cambiar por el otro). Si el padre es rojo, el hermano negro y los hijos negros, intercambiamos el color del padre con el del hermano. Así, todos los recorridos que pasan por el nodo hermano pasan por el mismo número de nodos negros, pero los recorridos que pasan por el nodo que se ha eliminado

tienen un nodo negro mas por el que pasar, consiguiendo así el número de nodos que se tenía antes de borrar el nodo negro y quedando equilibrado.

Si el primo cercano al nodo eliminado es rojo y el otro es negro, sin importar de qué color sea el padre, se realiza una rotación poniendo al primo de color rojo como el nuevo hermano e intercambiando el color de éste con el del hermano original. Esto se hace para dejar el árbol compatible con el último caso.

Si el color del primo lejano es de color rojo, sin importar de qué color sea el padre, se hace una rotación en el nodo padre, intercambiando su rol con el del hermano, e intercambiando a su vez sus colores. Por último se actualiza el color del nodo primo rojo a negro. Esto añade un nodo negro más a los recorridos que pasan por el nodo y el número de nodos negros de los recorridos que pasan por el primo que inicialmente era rojo, al cambiarlo a negro, también se mantiene.

El código Java asociado a toda esta explicación es el siguiente:

```
public RN elimina(RN rn, Comparable elem) {
        if (rn == null) return rn;
        else if (compara(rn.elemento, elem) < 0) {
        rn.hijoIzdo = elimina(rn.hijoIzdo, elem

        } else if (compara(rn.elemento, elem) > 0) {

        rn.hijoDerecho = elimina(rn.hijoDerecho, elem);
        } else {
            if (esHoja(rn)) {
              nodoBorrar=rn;
                   return rn;
            } else if (profundidad(rn.hijoDerecho) < profundidad(rn.hijoIzdo)) {
                rn = subeHijoIzdo(rn);
            } else rn = subeHijoDrcho(rn);

        }

        rn.altura = max(profundidad(rn.hijoIzdo),profundidad(rn.hijoDerecho)) + 1;
        if(nodoSustituido==rn)
              return nodoSustituto;
        else
              return rn;
}
```

*Nota: Por claridad, no se adjunta el código correspondiente a las funciones de equilibrado de los árboles.

Como viene siendo costumbre a lo largo de esta sección, vamos a tratar de ilustrar todo lo descrito de una forma más visual, con la intención de esclarecer un poco más todo lo comentado. Tomaremos un árbol concreto ejemplo e iremos explicando los distintos casos del proceso de equilibrado. Empezaremos borrando el elemento 10 del siguiente árbol:

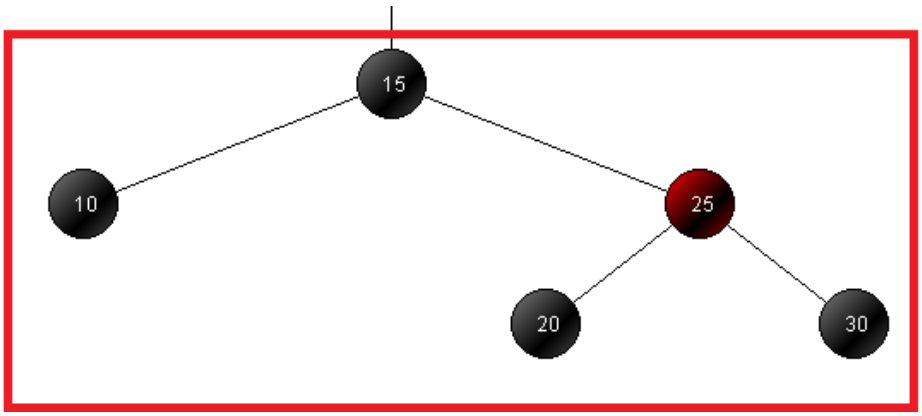

Figura 42. Árbol al cual eliminar el nodo 10

Para lograr equilibrar el árbol resultante necesitamos aplicar los siguientes pasos:

- Una rotación en el nodo 15 con el 25 y cambio de color entre ellos.
- Cambio de color en 15 a negro y en 20 a rojo.

El árbol resultante queda entonces:

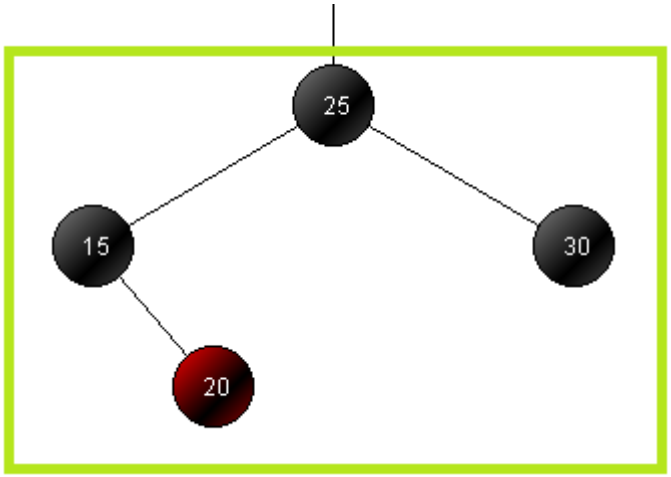

Figura 43. Árbol equilibrado tras una rotación y tres cambios de color

El siguiente paso será borrar el nodo 20 del siguiente árbol:

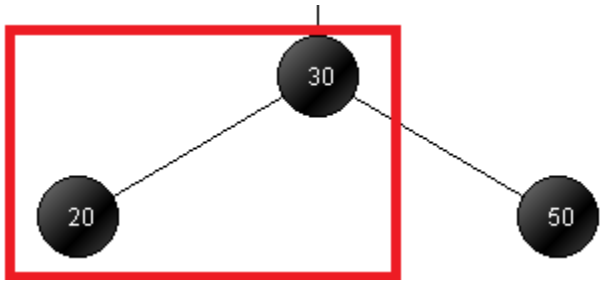

Figura 44. Árbol al cual eliminar el nodo 20

- Al eliminar 20, rompemos la condición del número de nodos negros entre raíz y hojas. Aplicamos un cambio de color sobre el nodo 50.

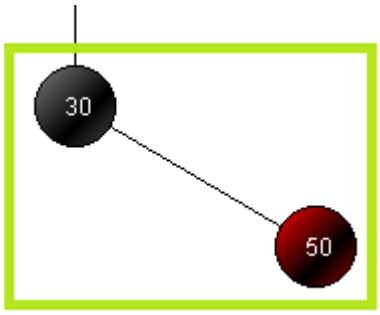

Figura 45. Árbol equilibrado tras un cambio de color

Como última ilustración vamos a atrevernos con un árbol un poquito más grande:

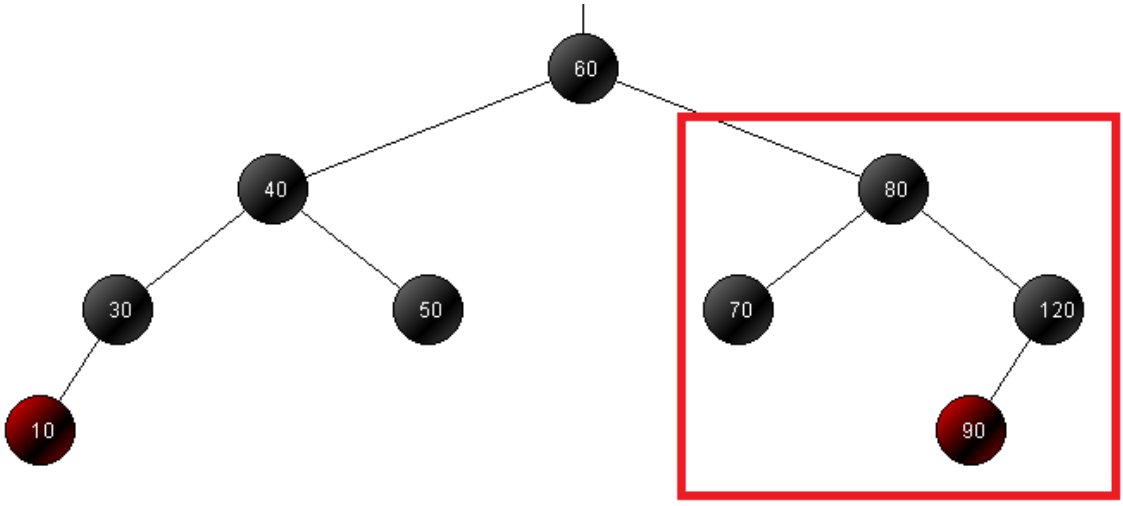

Figura 46. Árbol al cual eliminar el nodo 70

¿Qué ocurre si eliminamos el nodo 70?

- Se rota 120 con 90 (y se intercambian los colores) para que el árbol tenga un primo rojo derecho y no un primo rojo izquierdo.
- Se rota al padre 80 con el hermano (ahora 90 por la última rotación) y se intercambian los colores para equilibrarlo.

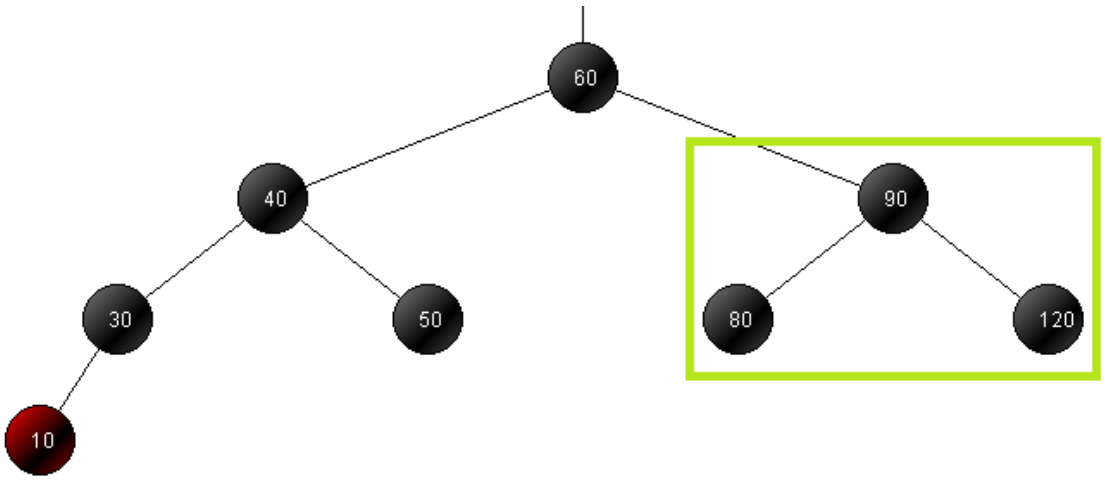

Figura 47. Árbol equilibrado tras dos rotaciones y dos cambios de color

Hemos terminado con todas las explicaciones asociadas a las estructuras de datos que maneja **V-Tree**, el siguiente paso será dilucidar un poco más el funcionamiento de las animaciones.

3.2.2.5. Animaciones

Para explicar su funcionamiento vamos a comenzar viendo la estrategia seguida desde un punto de vista muy abstracto, para ir concretando y profundizando en las diferentes partes hasta llegar al propio código.

La estrategia seguida para la animación es la siguiente. Cuando se realiza alguna operación sobre un árbol (ya sea eliminar o insertar), en la propia estructura se van acumulando las diferentes acciones que se deben animar.

Veamos un ejemplo de cómo se genera la secuencia de animaciones a realizar en el caso de inserción de un elemento, el 18 en el árbol de la figura. El algoritmo de inserción tomará las decisiones oportunas para llevar a cabo la colocación del nuevo elemento, esto lo aprovecharemos para fraccionar dicho algoritmo en conjuntos de instrucciones que representan una acción a animar. Más adelante describiremos con detalle el conjunto de acciones existentes.

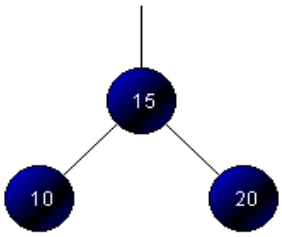

Figura 48. Árbol con secuencia de animaciones vacía y sin animar

En este instante, la secuencia de animaciones esta vacía.

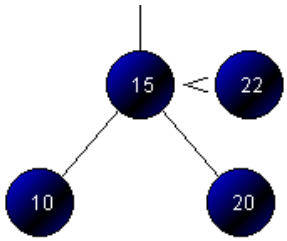

Figura 49. Árbol animando la acción "compara" tras haber animado "mueveRaiz"

En este momento, se han generado las acciones "mueveRaiz" y "compara"

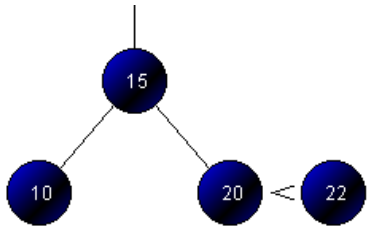

Figura 50. Árbol animando la acción "compara" tras haber animado "mueve"

Tras continuar por el hijo derecho, se habrán creado las acciones "mueve" y "compara".

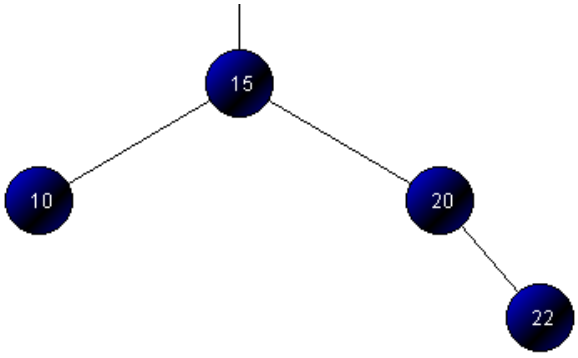

Figura 51. Árbol con secuencia de acciones vacía tras recibir "finAnimacion"

Por último, necesitamos incluir la acción "conecta". Marcaremos además el fin de la animación con la acción "finAnimación".

Una vez se tiene la secuencia de acciones generada por el algoritmo de inserción, se propaga a la clase animadora para que, paso a paso, vaya construyendo las animaciones asociadas a cada acción.

La clase animadora es una "máquina de estados" guiada por la secuencia de acciones generada. Determina qué animación realizar, y basándose en el estado actual del árbol y en unos parámetros, decide cómo animarlo. Para ello, posee una copia del árbol original con el cual va a ir fragmentando y conectándolo para dar como resultado cada uno de las transiciones que se tiene que mostrar por pantalla hasta llegar al árbol resultante.

Lo primero que veremos será cómo se ha ido guardando un registro de los pasos que el árbol original ha ido realizando para pasárselo a la clase animadora.

2.2.2.5.1. Pasos

Para poder representar las animaciones, hemos establecido una serie de acciones básicas que deben reflejar las animaciones, procurando siempre tener el menor conjunto de acciones posibles pero cubriendo todos los casos.

El conjunto final de acciones básicas es:

- ***mueve***: mueve un nodo auxiliar de un nodo origen del árbol a otro nodo destino.
- ***conecta***: conecta el nodo auxiliar como hijo del nodo origen.
- ***compara***: compara el valor de un nodo auxiliar con el valor del elemento del subárbol sobre el que se encuentra actualmente, pudiendo ser menor, mayor o igual.
- ***creaRaiz:*** crea el nodo raíz del árbol y lo coloca en la posición inicial.
- ***mueveRaiz:*** crea un nodo auxiliar y lo mueve al nodo raíz del árbol.
- ***rotaIzquierda:*** provoca una rotación a izquierdas respecto del nodo del árbol que se le indique.
- ***rotaDerecha***: produce una rotación a derechas respecto del nodo del árbol que se le indique.
  (Nótese que no se han introducido rotaciones dobles porque éstas en realidad se pueden generar combinando dos rotaciones simples.)
- ***cae:*** toma un nodo del árbol y lo desconecta del resto.
- ***subeIzquierdoSimple:*** respecto de la posición de un nodo, sube el subárbol de la izquierda y coloca al subárbol de la derecha como un hijo derecho del nodo raíz del subárbol izquierdo.
- ***subeDerechoSimple:*** respecto de la posición de un nodo, sube el subárbol de la derecha y coloca al subárbol de la izquierda como un hijo izquierdo del nodo raíz del subárbol derecho.
- ***sube:*** respecto de un nodo del árbol, sube su nodo sucesor o antecesor (el que se indique) a la posición que el nodo original ocupa.
- ***finAnimacion:*** indica que la animación se ha acabado y sirve para resetear e inicializar todas las variables globales que se han utilizado a los valores por defecto.

Para recopilar cada una de las acciones básicas que ha ido generando el algoritmo, el árbol tiene un atributo llamado “pasos”, que es una estructura de datos vectorial que almacenará todas las acciones provenientes de los algoritmos ya descritos.

La clase *Animacion* incluye el nombre de la acción básica a realizar, los elementos que se ven involucrados en el proceso (como puede ser un nodo origen y otro nodo destino) y por último el árbol resultante después del paso en caso de que se vea modificado.

Las constructoras de esta clase son:

```
public Animacion(String nombre, ArrayList<Comparable> parametros){
            this.nombre = nombre;
            this.params = parametros;
            this.arbol=null;
}
```

```
public Animacion(String nombre, ArrayList<Comparable> parametros,AVL
avl){
            this.nombre = nombre;
            this.params = parametros;
            this.arbol = avl;
}
```

Por último sólo nos queda ver cómo se rellenan estas acciones o pasos mientras se ejecuta cualquier instrucción que pueda realizar el árbol.

Para introducir los pasos, se han fraccionado los algoritmos de inserción y eliminación en bloques (conjuntos de instrucciones que constituyen una acción básica). Cada bloque genera una acción que se añade al atributo *pasos,* incluyendo además los parámetros necesarios para llevar a cabo las animaciones correspondientes. Sólo vamos a mostrar un caso sencillo como ejemplo. La instrucción *inserta:*

```
public AVL inserta(AVL avl,Comparable elem,ArrayList<Animacion> pasos){
      if (esVacio(avl)){
            avl.elemento = elem;
            parametros.add(elem);
            pasos.add(new Animacion("creaRaiz",parametros));
      }else{
            parametros.add(elem);
            parametros.add(avl.getElemento());
            pasos.add(new Animacion("mueveRaiz",parametros));
            avl = inserta(avl,elem,pasos);
      }
      pasos.add(new Animacion("finAnimacion",null));
      return avl;
}
```

Como vemos en el código, en caso de que el árbol AVL sea vacío creamos una nueva animación del tipo “creaRaiz”, que tomará como parámetros el elemento que debe crear como nodo raíz. Si no es vacío, incluimos la animación “mueveRaiz” y como parámetros el elemento actual con el cual crear un nodo auxiliar para animar y el elemento del árbol al que tiene que moverse. Posteriormente, se hace una llamada a la inserción recursiva *inserta*. En esta llamada se generarán las siguientes acciones a realizar y se seguirán incluyendo en el contenedor de acciones. Por último, siempre se introduce la acción “finAnimacion”, que indica que la animación asociada a la inserción ha terminado.

Antes de comenzar con la explicación sobre la implementación de las animaciones, vamos a detallar cómo se ha implementado la clase base sobre la que se ha extendido todo el funcionamiento: *RunPanel.*

2.2.2.5.2. RunPanel

En términos generales, es una clase que permite acoplar la funcionalidad de nuestras pizarras gráficas. Cada pizarra tiene un entorno de ejecución propio, que no influye en las demás (ni en el resto de la ejecución de la herramienta). Dicho de otro modo, son paneles editables que se ejecutan en un *thread* independiente.

La clase *RunPanel* es una extensión de la clase ya existente en Java *JPanel* a la cual se le ha añadido la funcionalidad de la interfaz *Runneable*. El funcionamiento básico de esta clase se basa en un bucle controlado por el atributo "isRunning". En este bucle se ejecuta el código que se implemente para cada una de las clases que extiendan a RunPanel. El atributo "*delay*" controla la cadencia de ejecución de este bucle.

Los atributos usados para esta clase son:

```
protected int frameRate = 10;
protected int delay = 20;
protected volatile boolean isRunning = false;
protected Thread animador = null;
```

Una vez se tiene un objeto de la clase *RunPanel*, al llamar a la función "iniciar()", al atributo "animador" se le asigna un thread, se inicia y se activa "isRunning" para que comience a ejecutarse.

```
public void iniciar() {
   if( animador == null || ! isRunning ) {
            animador = new Thread(this);
            isRunning = true;
            animador.start();
    }
}
```

Cuando el thread se ha iniciado, la clase ejecuta la función *run*, que realiza el bucle descrito anteriormente.

```
public void run() {
    while( isRunning )    {
        actualizarEstado();
        dibujarBuffer();
        dibujar();
        try  {
            animador.sleep(delay);
        } catch (Exception ex)  {
            ex.printStackTrace();
        } finally  {
        }
    }
}
```

Para controlar la velocidad a la que se re-ejecuta el código, tenemos la función "setFrameRate(int frameRate)", la cual modifica el atributo "frameRate" asignándole directamente el parámetro de entrada y actualiza el atributo "delay" en función de "frameRate" (y por consiguiente el retraso que se ejecuta entre vuelta y vuelta).

```
public void setFrameRate(int frameRate) {
    this.frameRate = frameRate;
    try  {
        delay = 1000 / frameRate ;
    } catch (Exception ex)  {
        ex.printStackTrace();
        delay = 20;
    } finally  {
    }
}
```

Una vez presentadas las ideas básicas de la implementación de las animaciones y especificada la forma de generar y propagar una secuencia de acciones, a continuación detallaremos cómo funciona la clase que implementa las animaciones.

2.2.2.5.3. VisualPanel

Esta clase extiende a la ya explicada RunPanel, de forma que posee toda la funcionalidad descrita y que además sobrecarga la implementación del método *run*.

Antes de comenzar vamos a ver las estructuras y atributos principales que hemos utilizado para la implementación de las animaciones. Por claridad, sólo explicaremos aquellos atributos con una funcionalidad crítica.

```
private AVLGraf<Comparable> arbol,arbolGraf;
```

Tenemos un atributo "arbol" en el cual guardamos en todo momento el árbol original lógico que se está usando para el programa general. Este árbol es la referencia de cómo tiene que acabar nuestro árbol después de cada instrucción (y de sus respectivas animaciones).

```
private AVLGraf arbol,arbolGraf;
```

El atributo “arbolGraf” contiene un árbol desactualizado un paso por detrás del árbol que se usa en el programa general. Este árbol se utiliza para poder ir representando todas las formas intermedias y animaciones que el árbol tiene que adaptar hasta llegar a ser igual que el original. Éste es el árbol que se estará dibujando en todo momento en la interfaz del usuario.

```
private HashMap<Comparable,Posicion> ElemPos=new HashMap<Comparable,Posicion>();
```

El HashMap “ElemPos” es una tabla que contiene como clave cada uno de los nodos del árbol y como valor un objeto de la clase “Posición”. Esta clase tiene dos atributos que indican una posición “X” y una posición “Y”. Esta tabla se utiliza para saber en todo momento la posición que un cierto nodo ocupa en la pantalla. Un uso muy evidente es saber dónde pintarlo.

```
private ArrayList<Comparable> Elems=new ArrayList<Comparable>();
```

El ArrayList “Elems” es una lista que contiene todos los elementos que están actualmente en el árbol. Es útil, por ejemplo, para poder recorrer todos y cada uno de los nodos de forma lineal. Esto es útil por ejemplo para poder recorrer los valores de la tabla “ElemPos”.

```
private int estado;
```

El atributo “estado” marca en todo momento lo que debe dibujarse en la interfaz o lo que se tiene que estar animando. Este atributo se modifica en función de la lista de animaciones que envía el árbol principal.

```
private float tiempoAni;
```

El valor real “tiempoAni” nos indica el tiempo que le queda a la animación actual antes de que acabe. Al comienzo de cualquier estado (excepto en el estado ‘0’) se inicializa este valor y cada cierto tiempo va bajando su valor hasta llegar a cero. También tiene otro uso clave en las animaciones marcando el ritmo al que tienen que avanzar.

```
private Animacion a;
```

El objeto “a” de tipo “Animación” va recogiendo cada una de las animaciones recibidas del atributo “pasos” del árbol original.

```
private boolean animando;
```

Este booleano indica si actualmente se está llevando a cabo alguna animación. Se utiliza para determinar si es posible comenzar la siguiente animación o si se debe esperar.

```
private Comparable origen,destino,elemAni,origen2,destino2,nodoParcialElem;
```

Esta serie de elementos contienen los valores de los nodos que tenemos que manipular para las animaciones.

```
private AVLGraf subArbol;
private HashMap<Comparable,Posicion> ElemPosSub=new
HashMap<Comparable,Posicion>();
private ArrayList<Comparable> ElemsSub=new ArrayList<Comparable>();
```

En esta clase hay hasta tres subárboles con sus respectivas tablas donde se indica la posición de cada uno de sus nodos. Estos subárboles se utilizan para poder seccionar el árbol principal y poder manipular cada parte por separado. Un ejemplo es la eliminación de un nodo (se generaría un subárbol por cada uno de sus dos hijos).

Las funciones principales que se han utilizado son las siguientes:

```
public void run()
public void paint(Graphics g)
```

Dada la importancia y complejidad de estas dos funciones, las explicaremos más adelante de una forma más extensa y detallada.

```
private void iniciaTabla()
```

Inicializa las tablas “ElemPos” y “Elems” en función del árbol principal. La función calcula cada una de las posiciones que debería ocupar cada nodo y lo almacena en la tabla “ElemPos”, además, va guardando en “Elems” cada uno de los valores del árbol. Normalmente se usa para acondicionar las posiciones de cada nodo, al cargar un árbol nuevo o después de cada animación.

```
private void iniciaTablaSub(…)
```

Esta función sirve para inicializar las tablas “ElemPosSub” y “ElemsSub” de uno de los subárboles. Como animar un subárbol implica calcular su posición en tiempo real, sería muy costoso tener que calcular la posición de cada nodo del subárbol en cada instante si éste es

muy grande. Por ello, a cada uno de los subárboles se les asigna una tabla de posiciones relativas a las que solo hay sumarles la posición del nodo raíz. Así, calcular la posición actual de cada nodo significa hallar la posición del nodo raíz y aplicar una sencilla suma para cada nodo hijo.

```
private void dibujaNodo(…)
```

Dadas una posición y un valor, dibuja un nodo en las coordenadas indicadas con el valor en el centro. Se utiliza generalmente para las animaciones de nodos.

```
private void pintaArbolR(…)
```

Esta función recursiva pinta un árbol en el panel dada una posición inicial para el nodo raíz. Esta función se utiliza, tanto para pintar el árbol original, como para pintar los subárboles en cada una de las posiciones intermedias por las que pasa cuando se está realizando alguna animación. Esta función hace uso de los atributos "ElemPos" y "Elems".

```
private void pintaArbol(…)
```

Esta función es una llamada inicial a la función recursiva "pintaArbolR(…)" con unas posiciones prefijadas. Posiciona el árbol principal en el centro del panel y lo pinta. Se utiliza para pintar el árbol principal cuando no se está llevando a cabo ninguna animación, o cuando se tiene que pintar un trozo del árbol principal de manera estática.

```
public void actualiza(…)
```

Esta función sirve para actualizar el árbol del entorno gráfico. Se le llama desde una clase externa y se le pasa el árbol que se quiere representar en el panel. Se suele utilizar, ya sea porque el árbol actual ha sido modificado o porque se ha creado uno nuevo. En caso de que el árbol contenga acciones de animación, el autómata se pone en marcha y realiza la serie de animaciones que sean pertinentes.

```
public void actualizaDirecto(…)
```

Este función realiza la misma tarea que *actualiza(…)*, con la diferencia de que cuando se actualiza el árbol mediante esta función, se omiten todas las posibles animaciones que se deberían llevar a cabo.

```
public void cargar(…)
```

Carga un árbol guardado, e inicializa las tablas necesarias.

Ahora explicaremos brevemente cómo funciona la máquina de estados sobre la que está basada toda la animación. Como ya hemos mencionado, la clase *VisualPanel* extiende de *RunPanel*, por lo que tiene una función *run()*. Ahí es donde hemos implementado todas las transiciones de la máquina de estados. En función de estos estados, es la función *paint(…)* la que pinta un árbol de cierta forma o realiza alguna animación. Primero vamos a ver como se producen los cambios de estado de la función "run()" y después cómo se ven reflejados éstos cambios en la función *paint(…)*.

Esquema del funcionamiento *de run():*

1. Cuando se inicia la máquina de estados, parte desde el estado '0'.
2. Cuando se está en este estado, se comprueba continuamente si hay alguna animación que realizar accediendo al atributo "pasos" del árbol principal.
3. Cuando desde fuera se llama a la función *actualiza(…)* el árbol principal es sincronizado con el árbol externo, normalmente con una serie de animaciones que realizar.
4. Cuando en el estado '0' se detecta que el atributo "pasos" tiene algún contenido, accede a la primera posición de "pasos" y almacena el contenedor de la animación a realizar en el atributo "a" de tipo *Animacion*. Una vez el objeto está a salvo en "a", éste se elimina de los pasos de "árbol".
5. Ahora se analiza el contenido de "a" y se determina el tipo de animación. Si la animación es del tipo "finAnimacion" se pasa al paso 8, en otro caso se vuelcan todos los elementos del contenedor a atributos de la clase, se inicializa el tiempo de la duración de la animación, y por último se cambia el atributo "estado" al estado correspondiente.
6. Una vez todo está preparado para ejecutar la animación, se pone la variable booleana "animando" a TRUE y se espera hasta que el tiempo de animación llegue a cero.
7. Cuando se ha consumido todo el tiempo de la animación, la variable booleana "animando" se vuelve a poner a FALSE y se vuelve a acceder a la primera posición de "pasos" del árbol principal para sacar el siguiente contenedor de animación en "a". Se elimina el objeto sacado de "pasos" y se pasa al paso 5.

8. En el este estado se actualiza “arbolGraf” a “arbol” y se inicializa la tabla de posiciones. Por último se pasa al estado ‘0’ y se vuelve al paso 2.

Para pintar cada una de las animaciones en la escena, se ha aprovechado la variable global “tiempoAni” para saber el porcentaje de animación que se tiene que haber completado en cada momento. Conociendo el porcentaje de la animación que se debería haber llevado a cabo en ese instante, podemos hallar la posición que debería ocupar cada uno de los elementos en el instante de pintarlos.

Por ejemplo, vamos a realizar la animación *mueve*. Esta animación hace que un nodo auxiliar se mueva desde un nodo origen del árbol a un nodo destino. Antes de comenzar la animación primero se guardan las posiciones del nodo origen y del nodo destino de la tabla “ElemPos”. Sabemos cuáles son estos nodos porque el objeto “a” de tipo “Animacion” contiene el valor del nodo origen y el del nodo destino. Cuando comienza la animación, el atributo “tiempoAni” va disminuyendo paulatinamente su valor. Ahora supongamos que tenemos que pintar la escena en el instante en el que la animación va por la mitad (“tiempoAni” está a la mitad de su valor original). Como tenemos la posición del nodo origen y la del nodo destino sólo habría que situar el nodo auxiliar en la posición central entre ambos nodos. Si el caso fuera que el tiempo ha bajado un tercio, habría que colocar el nodo auxiliar a un tercio de de la línea recta que une el nodo origen del nodo destino. Como se puede apreciar, la posición que debe tomar el nodo auxiliar es un punto intermedio entre el nodo origen y el nodo destino, a una distancia del nodo destino proporcional al tiempo que queda. Ésta es una de las animaciones más triviales que se han implementado en esta aplicación. Sin embargo, el resto, aunque más complejos, han seguido la misma lógica para representar cada uno de los puntos intermedios por los que pasa la animación.

Hay veces en las que una animación necesita movimientos más complejos que los de moverse recto respecto de una recta. Para este tipo de animaciones se subdivide el tiempo de animación y se crean pequeñas animaciones independientes que completan el recorrido que debería hacer la animación completa.

Esquema del funcionamiento de *paint()*:

1. Lee el estado actual en el que se encuentra para determinar qué animación se va a ejecutar (o para determinar qué animación se estaba ejecutando y seguir con ésta).

2. Pasa a la parte de animación que se le indica y muestra por pantalla un instante concreto según el parámetro de “tiempoAni” y de la animación actual.

3. Se vuelve al paso 1.

Estos son todos los estados que hay junto con sus correspondientes animaciones:

1. En este estado simplemente se muestra el árbol actual.

2. *mueve*

3. *compara*

4. *creaRaiz*
5. *mueveRaiz*
6. *rotaIzquierda*
7. *rotaDerecha*
8. *rotaIzquierdaDoble*
9. *rotaDerechaDoble*
10. *cae*
11. *subeIzquierdoSimple*
12. *subeDerechoSimple*
13. *sube*

Nota: en la versión final, tanto *rotaIzquierdaDoble* como *rotaDerechaDoble* han sido sustituidas por un par de ejecuciones de rotaciones simples.

3.2.2.6. Captura de imágenes

La función de captura de imágenes de **V-Tree** está implementada mediante un sencillo algoritmo que utiliza un objeto de tipo *Robot*. Este robot posee un amplio repertorio de llamadas al sistema que nos permite obtener la llamada *captureScreen.* Con esta llamada, tomamos el contenido del panel visual y lo volcamos a un buffer, que posteriormente nos permitirá crear la imagen en formato .png. Cada imagen generada tendrá como nombre por defecto la hora de creación de la misma y se guarda automáticamente en la ruta: ./src/imágenes/CapturasPantalla/

```
public class Captura{

static public void captureScreen(String fileName) throws Exception {
            int hora, minutos,segundos;
            Calendar calendario = new GregorianCalendar();
            hora =calendario.get(Calendar.HOUR_OF_DAY);
            minutos = calendario.get(Calendar.MINUTE);
            segundos = calendario.get(Calendar.SECOND);
            fileName = (fileName +hora+"-"+ minutos +"-"+segundos+ ".png");
            System.out.println(fileName);
            Rectangle screenRectangle = new Rectangle(150,80,1000,600);
            Robot robot = new Robot();
            BufferedImage image = robot.createScreenCapture(screenRectangle);
            ImageIO.write(image, "png", new File(fileName));
    }
  }
```

Todas las imágenes incluidas a lo largo de esta memoria que corresponden a estructuras de datos han sido creadas con esta funcionalidad. Nos ha permitido capturar imágenes en modo

ráfaga durante la ejecución de las animaciones, de forma que ha sido mucho más sencillo presentar visualmente los estados iniciales, intermedios y finales de un árbol sobre el que se ha realizado alguna acción.

3.2.2.7. Ejecución en escritura

El algoritmo principal de esta sección analiza un conjunto de instrucciones tomadas del panel de escritura y compone un árbol, AVL o rojo-negro, acorde a las instrucciones analizadas, pudiéndose observar el árbol resultante en el panel visual. Genera además, un fichero de texto que incluye anotaciones que indican cuándo y de qué tipo se producirá una rotación tras ejecutar una instrucción. Se trata de un sencillo intérprete de texto que procesa el conjunto de instrucciones hasta encontrar la palabra "fin" e identifica cada instrucción con los métodos *Insertar y Eliminar* de los árboles, ejecutando en cada caso el que corresponda con el valor indicado en cada instrucción. El código que resume este algoritmo es el que sigue:

```
while(!lineaLeida.equals("fin")) {

accion = extraerAccion(lineaLeida);
dato   = extraerDato(lineaLeida);

if(accion.contains("add"))
     avg = avg.inserta(avg.avl,dato);
else if(accion.contains("del"))
     avg = avg.elimina(avg.avl,dato);
else if(accion.contains("dibuja"))
     this.avl = avg.avl.clonar();

}
```

Como ejemplo vamos a ejecutar la secuencia de instrucciones del panel derecho hasta encontrar "draw" obteniendo el árbol del panel izquierdo:

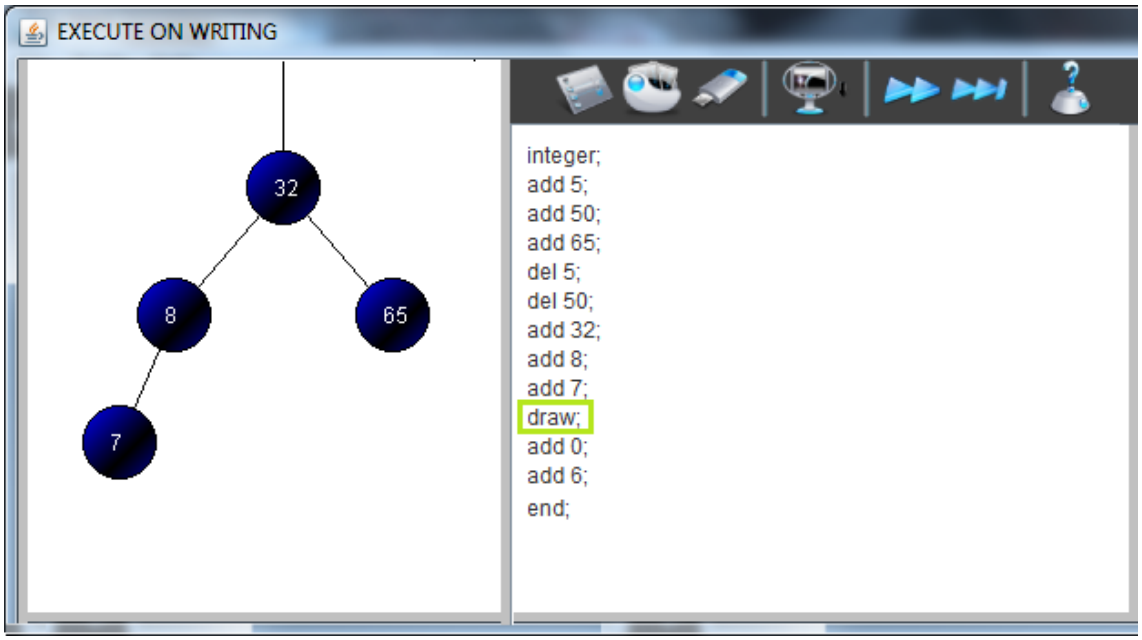

Figura 52. Ejecuta hasta "draw" y se muestra en la izquierda

Como segundo resultado, obtendremos la secuencia de instrucciones en el panel derecho y en ella se puede visualizar en qué puntos se producirán las rotaciones.

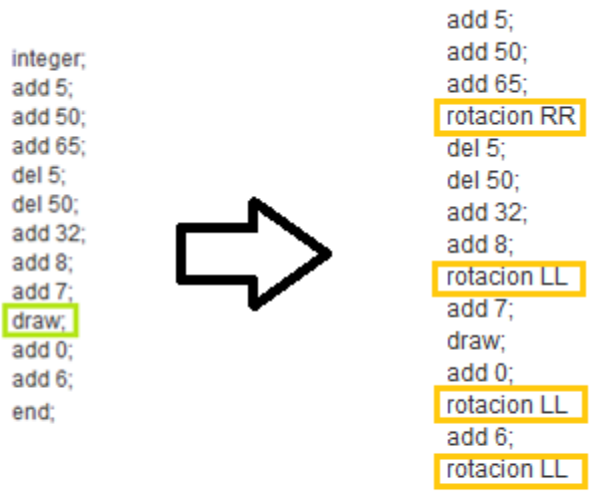

Figura 53. Resultado generado tras la ejecución

3.2.2.8. Test AVL-Rojinegro

Una de las novedades de esta herramienta es la inclusión de un apartado exclusivo para los tests y su personalización dependiendo del usuario que se haya registrado en la aplicación. De

esta manera, se aborda un tema que no se encuentra frecuentemente en estos tipos de herramientas gráficas de visualización de estructuras de datos arborescentes: la autoevaluación de los conocimientos adquiridos.

Partiendo de esta base, el desarrollo de los tests ha derivado hacia dos corrientes: el alumno y el profesor. Evidentemente, es contraproducente fijarnos solamente en el bien del alumno cuando éste trata un material que será generado y repuesto dinámicamente por una tercera persona. Si no facilitamos un entorno de fácil manejo para la gestión de los documentos, el propio profesor perderá el interés debido a la dificultad y el tedio de actualizar los materiales. Y ese desinterés se transferirá a sus estudiantes, desembocando en el desuso e inutilidad de la herramienta en su proceso de aprendizaje, habiendo sido en vano su creación.

En términos generales la mayor parte del trabajo ha sido idear una manera de mostrar los tests y almacenar las preguntas. Por este motivo nos valdremos de cuatro ficheros que funcionarán como pequeñas bases de datos: “baseEnunciadosAVL”, “baseEnunciadosRN”, “basePreguntasAVL” y “basePreguntasRN”. En las dos primeras almacenaremos los enunciados de las preguntas; en las segundas, escribimos las preguntas completas en el mismo orden en que se guardaron su enunciados en los otros archivos. Estas bases serán útiles para la gestión de los tests del profesor.

Ciertas clases son compartidas para la implementación de la parte de los tests de los alumnos y los maestros. Esas clases, que definiremos a continuación, son:

- **Pregunta:** Estructura que almacena todos los datos de una pregunta de la base de datos elegida. Los valores se le pasan por parámetros en la constructora. Proporciona una representación gráfica en un *JPanel* de la pregunta para que pueda mostrarse en los cuestionarios.

- **JPanelPreguntas:** Clase que se encarga de leer todas las preguntas del fichero de test que hayamos elegido. Cuando tiene todos los datos de una de ellas, le crea un objeto “Pregunta” y recoge su *JPanel* para unirlo al del resto de cuestiones en uno general. Añade también un apartado de comentarios y un botón para comprobar los resultados.

- **JPanelTest:** Devuelve un *JScrollPane* dentro del cual se encuentra el test completo generado por la clase anterior *JPanelPreguntas*. Así podremos visualizar enteramente el test gracias a las barras de desplazamiento.

Sin embargo, algunas otras pertenecen sólo a la parte programada destinada al uso del profesor.

- **JTreeNormal**: Clase que crea un *JTree* con los enunciados de las preguntas que componen nuestra base de enunciados.

- **JTreeCheck:** Clase que crea un *JTree* cuyos nodos son *JCheckNodes* y sus textos son cada uno de los enunciados de las preguntas que componen nuestra base. Es una clase

distinta a la anterior dado que este árbol está creado específicamente para la selección de preguntas, las cuales permitirán la creación de tests. Cuando se seleccionen los nodos ha de tenerse cuidado; hay que cliquear exactamente dentro del checknode. Si no es así, no se contará la pregunta internamente aunque aparezca seleccionada, y dará lugar a incongruencias e inconsistencias en el número de preguntas elegidas.

- **JPanelCreacionTest:** Genera un *JPanel* compuesto por un objeto de *JTreeCheck* y otro panel que contiene un objeto de la clase *JPanelColumnaLupas*. Así, aparecerá al lado derecho de cada enunciado del árbol una lupa que al presionarla, mostrará la pregunta al completo. Además, se podrán elegir qué cuestiones serán las elegidas para formar un test.

- **JPanelVisualizacionBBDD:** Genera un *JPanel* similar al facilitado por la clase anterior. En este caso, sólo se visualizarán los enunciados en cada nodo del árbol y se tendrá a la derecha la columna de lupas que permitirán visualizar las preguntas en su totalidad.

- **PreguntasLupa:** Estructura de datos formada por un botón cuyo icono es una lupa y un objeto de la clase "Pregunta". Se genera un panel que muestra una pregunta con su enunciado, imágenes, respuestas y con opciones de modificar o eliminar. Dicho panel sólo se verá si se pulsa el botón lupa.

- **JPanelColumnaLupas:** *JPanel* formado por una columna de lupas contenedoras de las preguntas de la base de datos que se usa en ese momento.

- **VentanaEliminarTestProf:** Pequeña ventana auxiliar que permite elegir de entre los tests actuales cuál es el que se quiere borrar.

- **VentanaCreacionPregunta:** Proporciona una ventana que permite completar los campos necesarios para introducir una nueva pregunta en el sistema, la cual se añadirá a la actual base de datos y será accesible desde ese mismo momento. Es posible acceder a esta ventana desde los botones lupa que aparecerán a la derecha de los árboles de las clases *JPanelCreacionTest* y *JPanelVisualizacionBBDD* comentadas anteriormente: en dicho caso, mostrará en cada apartado los datos correspondientes de la pregunta seleccionada, permitiendo que sean modificados ampliamente.

El planteamiento de la implementación para el profesor ha sido más complejo que el relativo al alumno, debido a las múltiples opciones que presenta y a la utilización de una base de preguntas. Esas actividades merecen ser comentadas para un conocimiento más profundo del funcionamiento de esta parte.

Al mismo tiempo que se genera la ventana del profesor, se almacenarán en el programa en una estructura de *ArrayList<Pregunta>* todas las cuestiones leídas de un fichero nombrado o bien "basePreguntasAVL" o "basePreguntasRN". Así pues, todas las modificaciones, eliminaciones y creaciones de preguntas que tengan lugar a lo largo de la aplicación en el apartado de test seleccionado, modificarán este *ArrayList<Pregunta>.* Una vez se cierre la

aplicación, o cambiemos de actividad dentro de la propia herramienta, todos los datos del ArrayList<Pregunta> serán reescritos en el mismo fichero que leyó en un principio. Esto permitirá mantener la consistencia de los datos con los nuevos cambios realizados.

- **Watch test**: Mediante la elección de uno de los niveles disponibles de la ventana auxiliar que surgirá inicialmente, se mostrará por pantalla el test elegido. Se podrá rellenar y ver cuales son las respuestas correctas. Además, todos los tests tendrán una carpeta asociada con las imágenes correspondientes al test; así se recurrirá a ellas de manera general y se evitará tener que acudir a destinos diferentes.

- **Create Test:** Aparecerán en la pantalla todas las preguntas que formen la base de preguntas del tipo de estructura de la cual hayamos seleccionado el test. Formarán parte de un *JTree* en el cual tendremos que seleccionar diez casillas y pulsar el botón "Create test" en la parte inferior para generar nuestro nuevo test. Para nombrarlo, el sistema comprobará cuantos cuestionarios existen: si el último era el "4", procederá a nombrar el nuevo como "5" y creará una carpeta llamada "A5" o "R5", en donde copiará y almacenará todas las imágenes incluídas en el test. Así, aunque borremos una pregunta de la base en el futuro, el test seguirá teniendo sus recursos intactos.

- **Delete Test**: Cuando creamos un test, su nombre coincide con la posición que ocupan: el primer test se llama "1", el segundo "2", etc. Así pues, al eliminar uno de ellos, si existen cuestionarios posteriores a él, su nombre será asumido por el siguiente, y así repetidamente, hasta que el nombre del último de ellos coincida con el número de tests existentes actualmente. También se borrará la carpeta de imágenes asociada al test eliminado, y el resto de carpetas de imágenes posteriores, si existen, también asumirán un cambio de nombre, hasta que la última de ellas se llame igual que el número total de cuestionarios.

- **Watch BBDD**: Muestra un árbol con todas y cada una de las preguntas de la base de datos. Al igual que en el árbol generado en "Create Test", al lado derecho aparecerá una columna de botones con imágenes de lupa. Cada uno de estos botones se relacionará con uno de los enunciados del árbol, y mostrará la pregunta al completo, permitiendo a su vez que dicha cuestión pueda ser modificada o eliminada.

- **New Question**: Interfaz que nos permitirá introducir todos los datos necesarios para generar una nueva pregunta que se añadirá a la base de preguntas. También servirá para mostrar los datos de un pregunta previamente seleccionada y modificarla, en cuyo caso dichos datos también se verán reflejados en la base.

- **Register Pupils**: Aparecerá una pantalla en la que introduciremos los correos de todos y cada uno de los alumnos del profesor. Cuando se termine y se acepte, se mandará a cada estudiante una contraseña generada aleatoriamente que tendrá que usar junto a su email para entrar en el aplicación. Al no haberse podido colocar por el momento esta herramienta *on-line*, la única solución es que el profesor dé de alta a todos los interesados antes de que ellos se descarguen la herramienta.

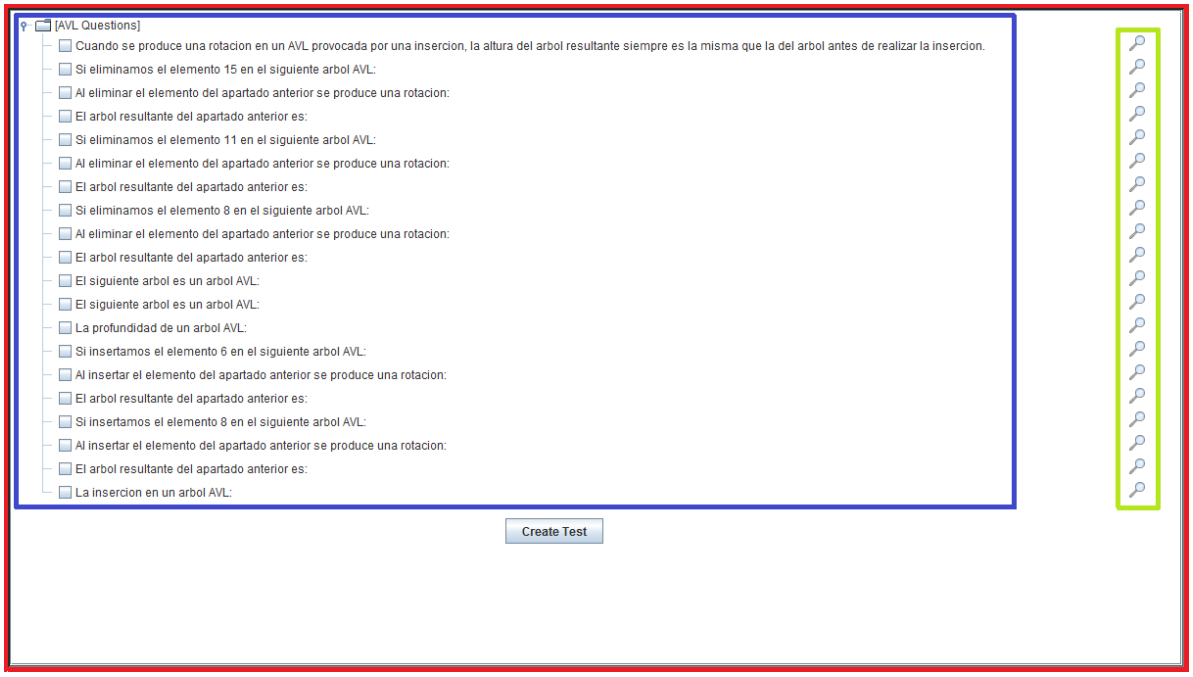

Figura 54. Panel de creación de Test

➤JTreeCheck ➤JPanelColumnaLupas ➤JPanelCreaccionTest

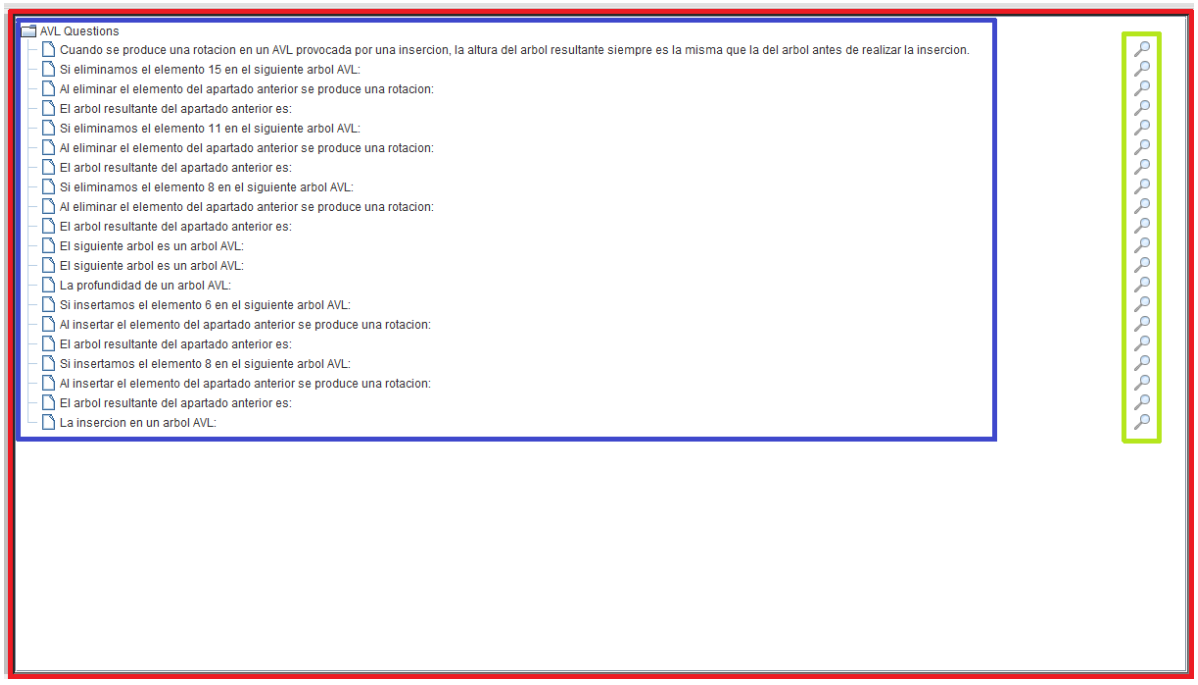

Figura 55. Panel de visualización de la base de datos

➤JTreeNormal ➤JPanelColumnaLupas ➤JPanelVisualizacionBBDD

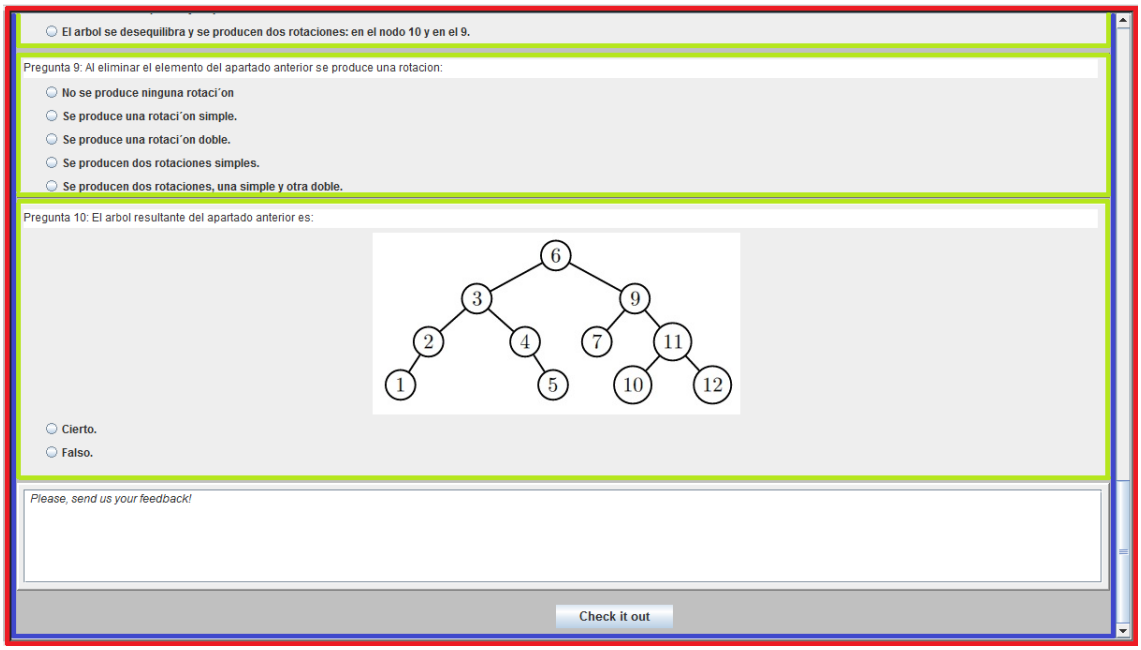

Figura 56. Panel de ver Test

➤JPanelPreguntasTest ➤Pregunta ➤JPanelTest

3.2.2.9. Interfaces

El objetivo principal de las interfaces ha sido proporcionar un entorno gráfico agradable e intuitivo de fácil manejo que cubra todas las necesidades del usuario, al mismo tiempo que no suponga una gran carga de mantenimiento para la propia aplicación. Nuestro objetivo principal es la funcionalidad, por lo que unos entornos sencillos e intuitivos centrarán la atención, dependiendo del usuario, en el aprendizaje o la gestión.

Consultamos la herramienta Vedya desarrollada por esta facultad. Estudiamos su presentación, su distribución de código y su ejecución. Como resultado observamos que existían problemas de repintado de las interfaces y diálogos que provocaban dudas sobre su finalidad. Así pues, decidimos partir de cero en la programación de la parte gráfica, basándonos parcialmente en la distribución estructural de algunas secciones como la ventana de inicio de la aplicación y las pertenecientes a la parte práctica de AVL y Rojinegro.

Las imágenes utilizadas para el fondo de pantalla inicial, botones, tests, han sido obtenidas de las empleadas en la herramienta Vedya o VedyaTest . Sin embargo, algunos iconos han sido conseguidos a través de la web al ser de libre propiedad.

De la herramienta Vedya hemos empleado con algunas modificaciones, el código del control de velocidad de las animaciones y los hilos para la animación de entrada de los botones en la interfaz inicial.

Para un par de casos, se ha consultado y reusado código hallado en internet [4], siendo estos los correspondientes a las clases:

- **CheckBoxNode, CheckBoxNodeEditor, CheckBoxNodeRenderer y NamedVector:** Usadas en la clase *JTreeCheck* para poder hacer un árbol cuyos nodos sean *JCheckBox* dentro del apartado test del usuario profesor. El código original en: http://www.java2s.com/Code/Java/Swing-JFC/CheckBoxNodeTreeSample.htm

- **Captura pantalla**: Obtiene una imagen rectángular de la pantalla actual, habiendo definido previamente el tamaño del rectángulo. Original encontrado en el blog: http://javalangnullpointer.wordpress.com/2007/03/30/capturar-la-pantalla-con-java-screen-capture/

Antes de explicar la interconexión entre interfaces y su relación, enumeraremos y aclararemos las clases principales que participan en el proceso.

- **VentanaInicial:** Clase que genera el primer panel que aparecerá al ejecutar el .jar de la aplicación. En ella el usuario se registrará; si ha tenido éxito elegirá entre manipular AVL's, RN's o test de AVL o RN. Se pueden ver los créditos también.

- **VentanaControl**: Interfaz principal que define un entorno genérico para el manejo de los árboles AVL o RN dependiendo de cual haya sido la opción seleccionada. Está formada por objetos de las clases *BarraHerramientas*, *BarraIconos*, *MarcoControl*, *JPanelPestagnasControl* y algunos otros elementos como *JPanel's*. A través de ella se seleccionarán las acciones a llevar a cabo, y se mostrarán los resultados finales.

- **BarraHerramientas***: JMenuBar* personalizado con todas las opciones necesarias para la gestión. Ocupa la posición más alta del *JFrame* proporcionado por *VentanaControl*.

- **BarraIconos***: JPanel* formado por varios *JButton's* y un control de velocidad sobre la animación de las animaciones. Debajo de la barra de herramientas.

- **MarcoControl:** *JPanel* que recoge todas las funciones aplicables sobre los árboles AVL y RN. Contiene un pequeño cuadro en la parte inferior que muestra el número de rotaciones de cada tipo que se han hecho hasta el momento, y que es susceptible de modificarse en cada aplicación de las funciones *Add* y *Delete*. Se sitúa en la parte izquierda de la pantalla.

- **JPanelPestagnasControl**: *JTabbedPane* que puede contener hasta 10 pestañas de árboles. A su vez cada pestaña está formada por un *JPanel* blanco en donde se representa y modifica el AVL o RN, y otro panel en el lado derecho que mostrará los pasos que se han dado hasta el momento. En dicho panel, será posible volver a pasos anteriores al pulsar los botones de la modificación llevada a cabo en aquel momento.

- **SingletonControl:** Clase que tiene como atributos objetos de las clases *OpcionInicial* y *VentanaControl*. Será invocada desde otras clases para que realice inmediatamente cambios en *VentanaControl*, como crear un árbol, eliminar alguno de sus nodos, recorrerlo, etc. Sirve como puente de comunicación y disparador de eventos entre varias clases.

- **VentanaTestAlumno**: Entorno genérico para los alumnos y sus test de AVL y RN. Antes de acceder a la pantalla, tendremos que elegir el nivel que deseamos hacer. Estará compuesta por objetos de la clases *BarraHerramientasTest*, *MarcoTestAlumno* y *JPanelTest*, además de *JFrame y JPanel*.

- **BarraHerramientasTest*:*** *JMenuBar* personalizado para las necesidades de la zona de los tests. Se utiliza la misma para los usuarios alumno y profesor.

- **MarcoTestAlumno**: *JPanel* que contiene un pequeño "canvas" blanco para simular un pequeño árbol AVL o RN dependiendo del test que se haya elegido. Sólo se puede crear, añadir nodos y eliminarlos: funcionalidad básica, ya que esta parte es para la autoevaluación.

- **JPanelTest*:*** *JPanel* que muestra las diez preguntas obligatorias del test elegido y una extra para enviar comentarios. Al final del mismo hay un botón "confirmar" para comprobar si las respuestas son las correctas.

- **VentanaTestProfesor**: Entorno gráfico genérico para la gestión de los test AVL y RN. Posee atributos de las clases *BarraHerramientasTest*, *MarcoTestProfesor*, *JPanelTest*, *JPanelCreacionTest* y *JPanelVisualizacionBBDD*.

- **MarcoTestProfesor**: *JPanel* contenedor de las funciones útiles para un profesor a la hora de gestionar la parte teórica de la aplicación, y el registro de los alumnos.

- **JPanelCreacionTest:** *JPanel* con un atributo de la clase *JTreeCheck*. Usado para crear un nuevo test, mostrará un árbol con *JCheckBox* por nodos de los cuales 10 tendrán que ser seleccionados. A su derecha, cada nodo tendrá el icono de una lupa para mostrar la pregunta al completo y poder optar a modificarla o eliminarla.

- **JPanelVisualizacionBBDD**: *JPanel* con un atributo de la clase *JTreeNormal*. Devolverá un panel con un *JTree* en cuyos nodos aparecerá una de las preguntas de la base de datos. Al igual que en el caso anterior, cada nodo tendrá el icono de una lupa a su

derecha.

- **SingletonTest:** Clase que actuará, como *SingletonControl*, de disparador de eventos o puente entre clases como *MarcoTestProfesor* y *BarraHerramientasTest* para que las acciones pulsadas en ellas se reflejen en *VentanaTestProfesor*. En el caso de *VentanaTestAlumno*, solamente se encarga de su inicialización, su eliminación y su cambio por una de las otras tres opciones del programa.

A continuación, mostraremos el diagrama de clases de las interfaces de la aplicación. Lo hemos dividido en dos partes. Una parte de control, que sería la práctica, es decir, la empleada en la creación y manipulación de árboles AVL y Rojinegros; y otra de tests, que sería la teórica, en la que se ponen a prueba los conocimientos o se gestionan los cuestionarios.

PARTE PRÁCTICA: Árboles AVL y Rojinegros

En esta sección, el alumno y el profesor acceden a la misma sección en las mismas condiciones: misma funcionalidad, mismos límites.

Al ejecutar el acceso directo de nuestra aplicación, o en su defecto el .jar, aparecerá en pantalla la interfaz de la *VentanaInicial*. Tras registrarnos con éxito, pulsamos el botón con la opción "AVL". Se llamará a *SingletonControl* para almacenar la opción elegida, es decir, si queremos manejar árboles AVL o Rojinegros; e iniciaremos la *VentanaControl*, creando todos sus elementos. La clase *SingletonTest* también se invocará para guardar el tipo de usuario que ha entrado en la aplicación, dado que es posible que más tarde el cliente desee cambiar de actividad.

A continuación se nos habrá abierto el *JFrame* proporcionado por la clase *VentanaControl*. Cualquier botón que pulsemos en su barra de herramientas, de iconos o en los paneles avisará a *SingletonControl*, quien se ocupará de pasar el mensaje a *VentanaControl* para que se ejecuten en su interfaz los cambios pertinentes.

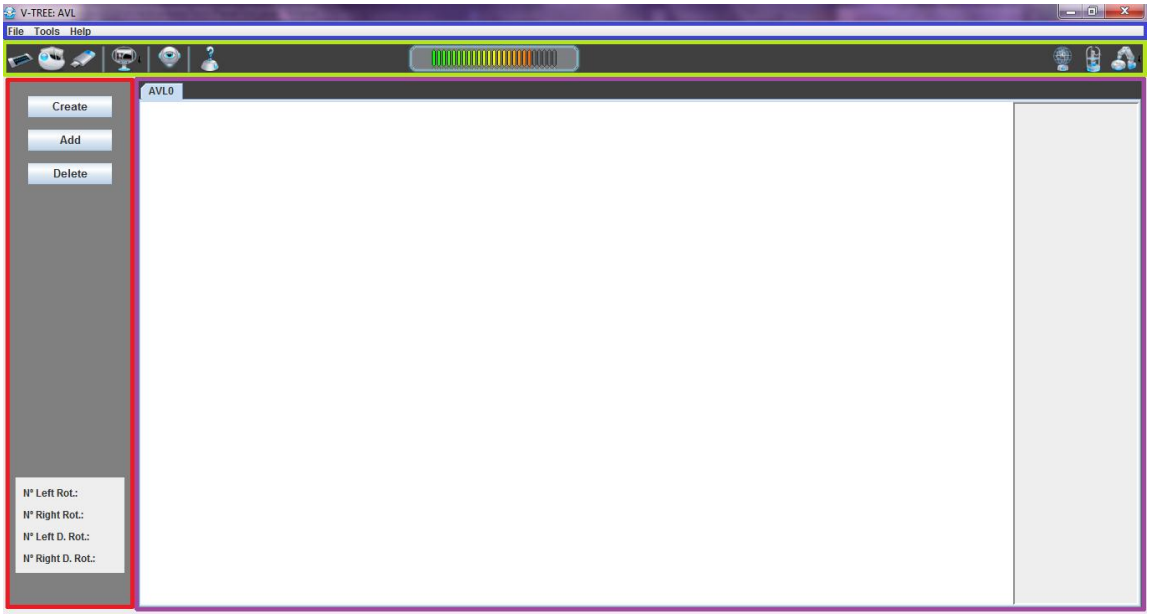

Figura 57. Interfaz de la parte práctica

➤BarraHerramientas ➤BarraIconos ➤MarcoControl ➤JPanelPestagnasControl

PARTE TEÓRICA: Test AVL y Rojinegros

Si el usuario se ha registrado como alumno, tras presionar la opción de "Test AVL" o "Test Rojinegros", *SingletonTest* iniciará la clase *VentanaTestAlumno* y establecerá las opciones de usuario e inicial. Además de esta acción, *SingletonTest* intervendrá en la manipulación del árbol que se podrá manejar en el margen izquierdo, para cambiar de nivel de test y en la gestión de cambio de actividad en caso de que el usuario quiera realizar otro tipo de test o pasar a la parte práctica.

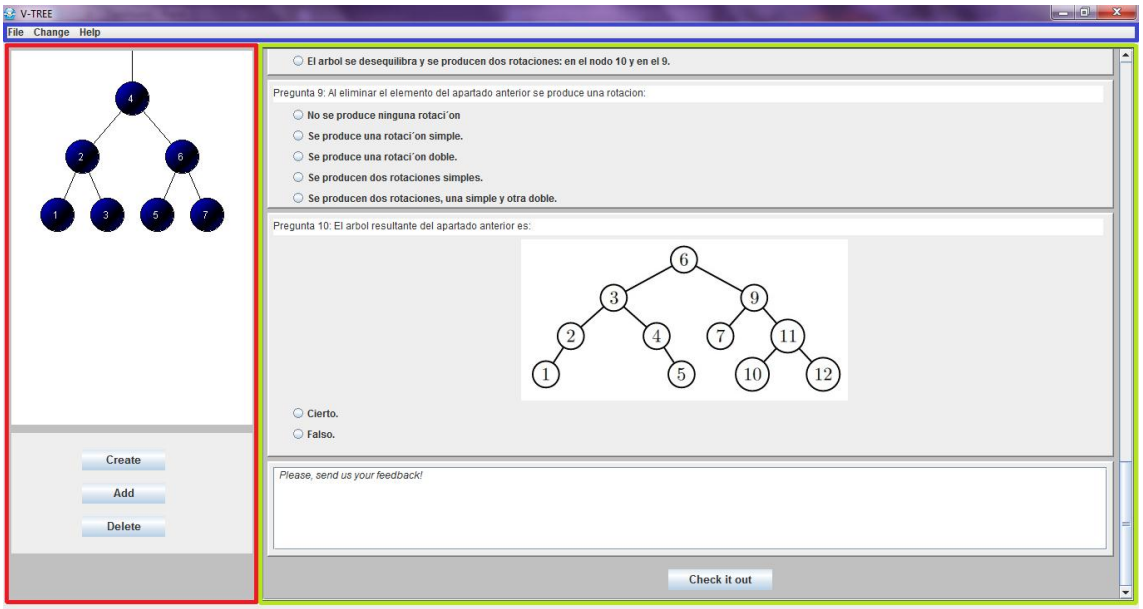

Figura 58. Interfaz de la parte teórica en modo alumno

➤BarraHerramientasTest ➤MarcoTestAlumno ➤JPanelTest

Si el usuario es profesor, *SingletonTest* dará la orden de iniciar la clase *VentanaTestProfesor* y almacenará la opción de usuario y la inicial elegida (por ejemplo, si trataremos sobre AVL's o Rojinegros). Paralelamente a lo que ocurría con *SingletonControl, SingletonTest* comunicará a *VentanaTestProfesor* qué modificaciones deben tener lugar en su interfaz según la función que haya sido activada a través de alguno de sus componentes.

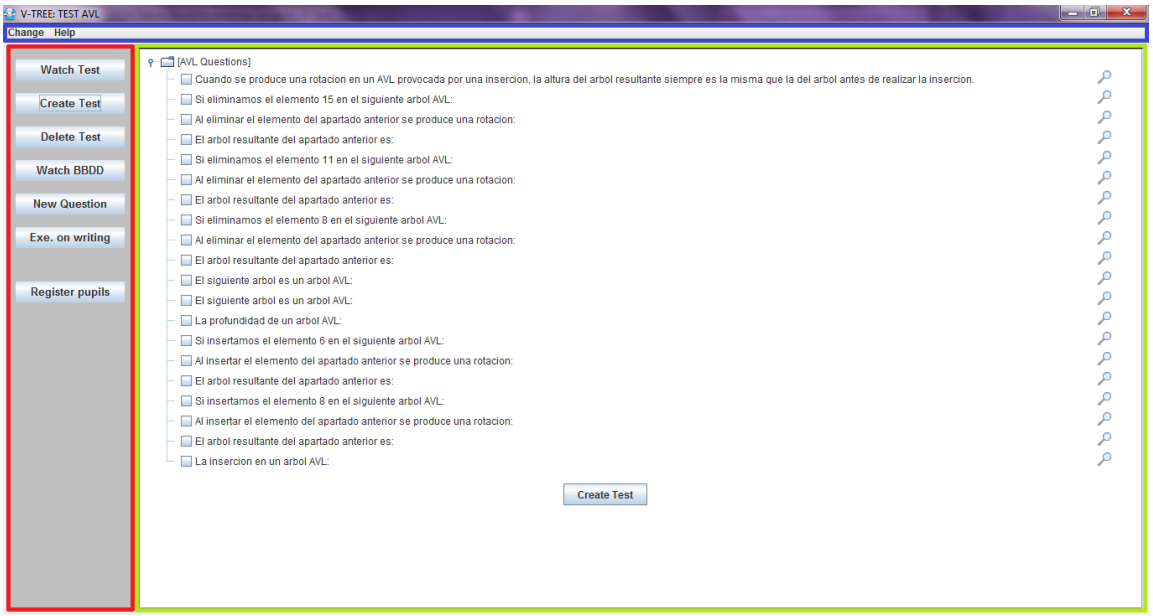

Figura 59. Interfaz de la parte teórica en modo profesor

➤BarraHerramientasTest ➤MarcoTestProfesor ➤JPanelTest

3.3. Despliegue del proyecto

V-Tree es una herramienta destinada a utilizarse *on-line*, en concreto en la plataforma Moodle del Campus Virtual [5] para que ésta sea accesible a todos los integrantes de un grupo. El profesor dará de alta a todos sus alumnos inscritos en la asignatura y éstos podrán utilizarla libremente después de su registro. Actualmente, aún no ha sido probada en este ámbito por distintas circunstancias, por lo que este prototipo sólo puede probarse vía ejecutable .jar.

3.4. Manual de Uso

Esta sección ofrece una pequeña guía de uso de la herramienta. A continuación se explicará con detalle cada sección de la aplicación así como las posibilidades y forma de uso de cada una de ellas.

3.4.1. Login

Antes de comenzar a utilizar **V-Tree**, tenemos que estar registrados en la base de datos, ya sea como alumno o como profesor. Sin esto, no podremos acceder a ninguna funcionalidad de la herramienta. Así pues, el primer paso deberá darlo el profesor registrándose en la aplicación para poder dar de alta a su vez a los alumnos que accederán a la herramienta.

En la interfaz inicial, el docente pulsará sobre el botón circular amarillo con el dibujo de la interrogación.

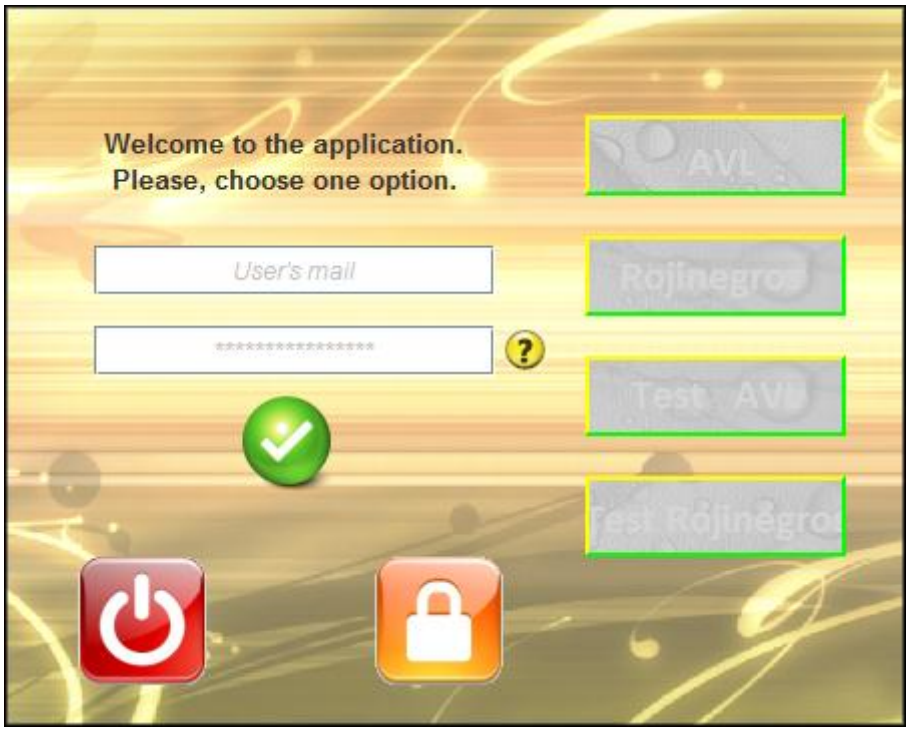

Figura 60. Ventana inicial

A continuación aparecerá una pequeña ventana en la cual el profesor deberá introducir su e-mail universitario de la UCM, cuya terminación será distinta de “estumail.ucm.es” aunque sí finalizará por “ucm.es”.

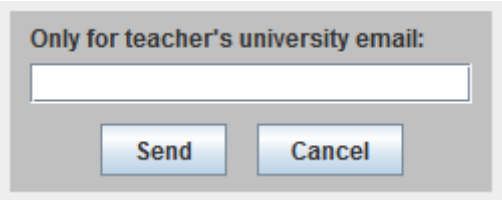

Figura 61. Ventana donde se introduce el correo del profesor

Un mensaje de bienvenida junto con una contraseña personal generada aleatoriamente será enviada al correo del maestro desde la propia cuenta electrónica de la aplicación: v.tree.app@gmail.com.

Si el proceso de registro ha sido válido, aparecerá un mensaje en pantalla informando del envío del correo, y el nuevo usuario docente será almacenado en un registro de la herramienta parametrizado por los campos: *usuario, contraseña, tipo de usuario*.

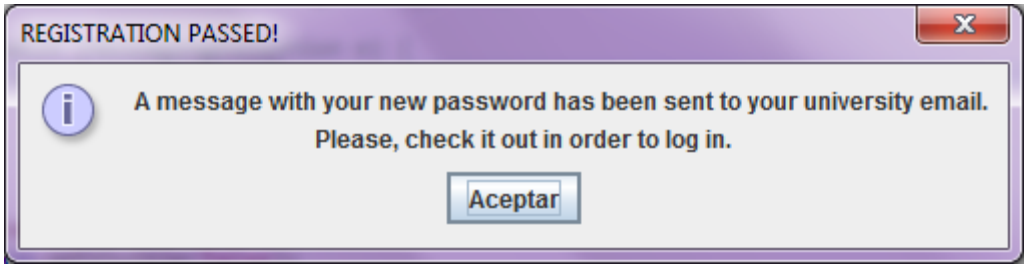

Figura 62. Ventana que indica que la cuenta se ha creado correctamente

Una vez el profesor o los alumnos hayan sido registrados, podrán introducir su correo universitario como usuario, y la clave que haya sido enviada a sus correos como contraseña. Al pulsar entonces sobre el botón circular verde de confirmación, las cuatro opciones de la derecha serán desbloqueadas para su posible selección.

Figura 63. Ventana inicial tras iniciar sesión

Existen dos contraseñas genéricas para el acceso de los alumnos o profesores a modo de prueba, siendo el nombre de los usuarios “alumno” o “profesor” respectivamente y “123abc” la contraseña.

En caso de que el usuario introducido no se encuentre en la base de datos de gente registrada, se comunicará al cliente con el siguiente mensaje.

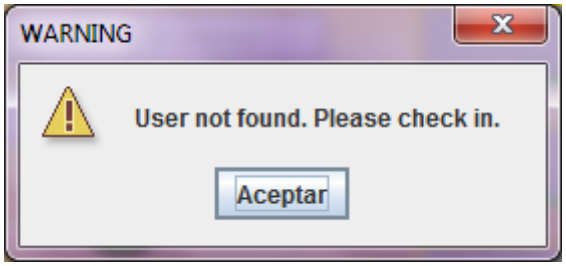

Figura 64. Aviso se usuario no existente

En caso de que el problema sea una contraseña inválida, no coincidente con la que se tiene almacenada, se mostrará la advertencia.

Figura 65. Aviso de contraseña errónea

El botón cuadrado rojo en la parte inferior de la pantalla servirá para salir de la aplicación, mientras que el naranja con el candado blanco mostrará los créditos de la aplicación, es decir, una breve información sobre los autores de la misma.

3.4.2. AVL y Rojinegro

Desde la interfaz inicial y una vez hecho login, pulsamos el botón “AVL” o “Rojinegros”, botones primero y segundo de la columna de la derecha respectivamente. Tras esto, aparecerá una pizarra sobre la que podremos empezar a realizar nuestras construcciones. Supongamos, por ejemplo, que escogimos la opción “Rojinegro”.

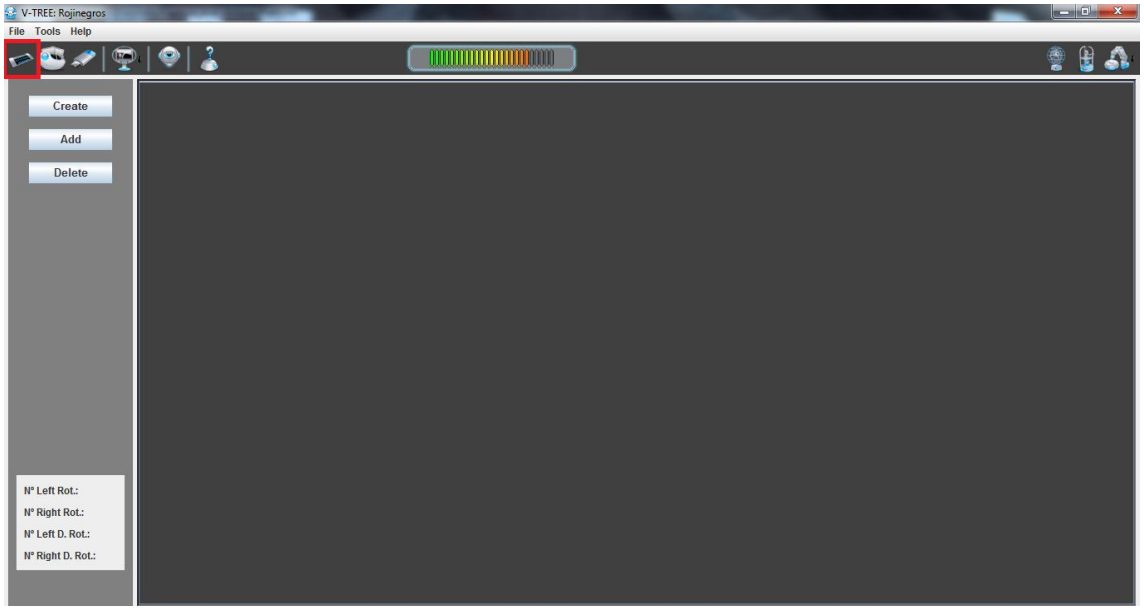

Figura 66. Aspecto inicial del modo práctico

Inicialmente, la pizarra estará de color negro y no será posible construir ninguna estructura. Para comenzar, debemos pulsar en el primer icono de la barra de herramientas "New Page". Ya podemos comenzar a dibujar en nuestra pizarra. Podemos además crear un máximo de diez páginas si queremos trabajar con varias estructuras a la vez.

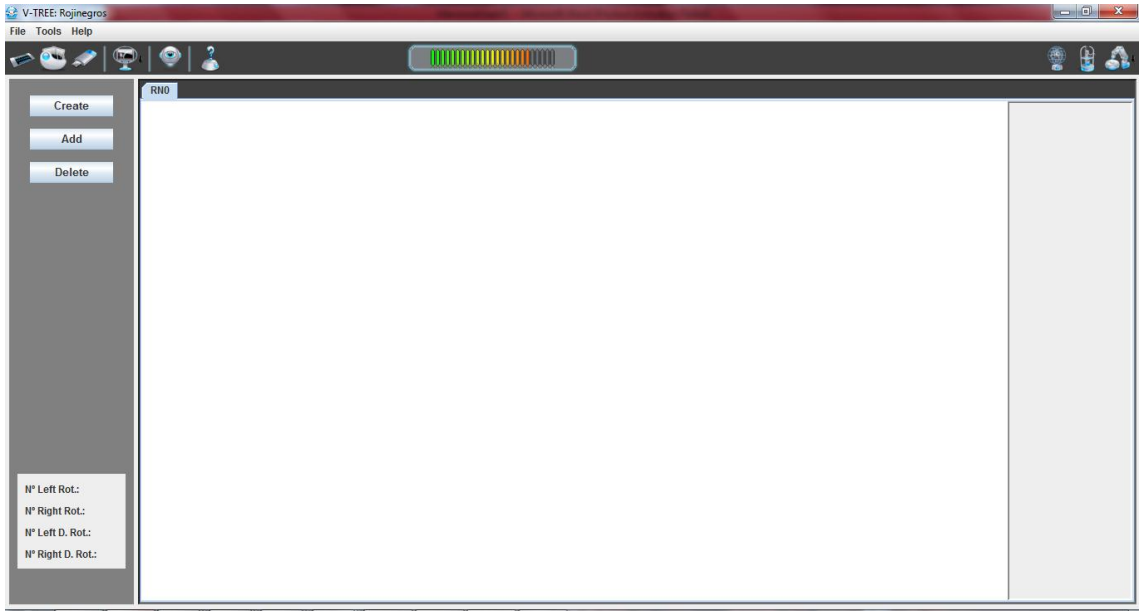

Figura 67. Aspecto tras crear una pestaña

A continuación, podemos crear una nueva estructura pulsando en el botón "Create" o podemos cargar una guardada anteriormente. Para ello, pulsamos en la pestaña File -> Open, la ruta corta CTRL+O o en el segundo icono de la barra de herramientas y escogemos el archivo que queramos visualizar. Si optamos por crearlo, tendremos que especificar el tipo de los datos que tendrá nuestra estructura. El siguiente paso es comenzar a trabajar sobre nuestra

estructura, si pulsamos el botón “Add” se nos pedirá el valor del elemento que queremos añadir y lo mismo si pulsamos “Delete” pero esta vez para eliminarlo.

Cada vez que realizamos una operación, aparecerá un nuevo botón en la columna de la derecha de la pizarra. Estos botones nos permitirán regresar o avanzar hacia estados anteriores/posteriores, pulsando sobre ellos, cambiaremos de estado.

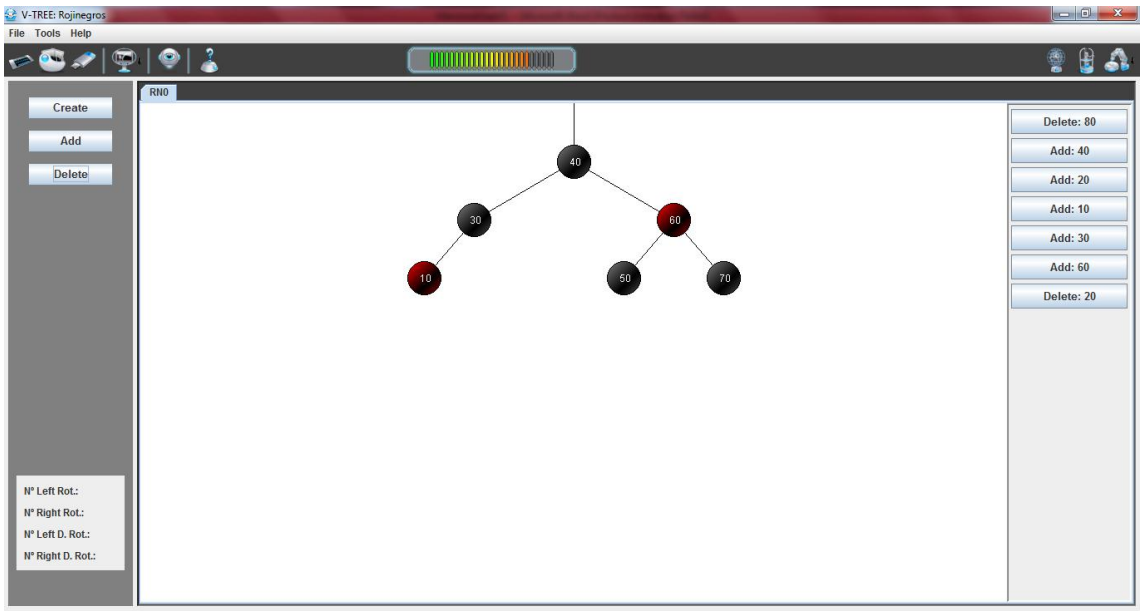

Figura 68. Aspecto mientras se usa

Podemos decidir guardar una estructura creada, para ello, pulsamos en la pestaña File -> Guardar, la ruta corta CTRL+G o el tercer icono de la barra de herramientas, damos un nombre al fichero y confirmamos.

Podemos modificar la velocidad de las animaciones aumentando o disminuyendo la carga de la barra de colores que hay sobre la pizarra. ¡No olvides aumentarla si quieres crear estructuras de forma rápida!

Para realizar capturas de imágenes sólo debemos pulsar en el cuarto icono de la barra de herramienta. Nuestra imagen se encontrará almacenada en el directorio ./src/imágenes/CapturasPantalla. Pulsando en el quinto icono de la barra, accedemos al modo Ejecución en Escritura, que pasamos a describir a continuación.

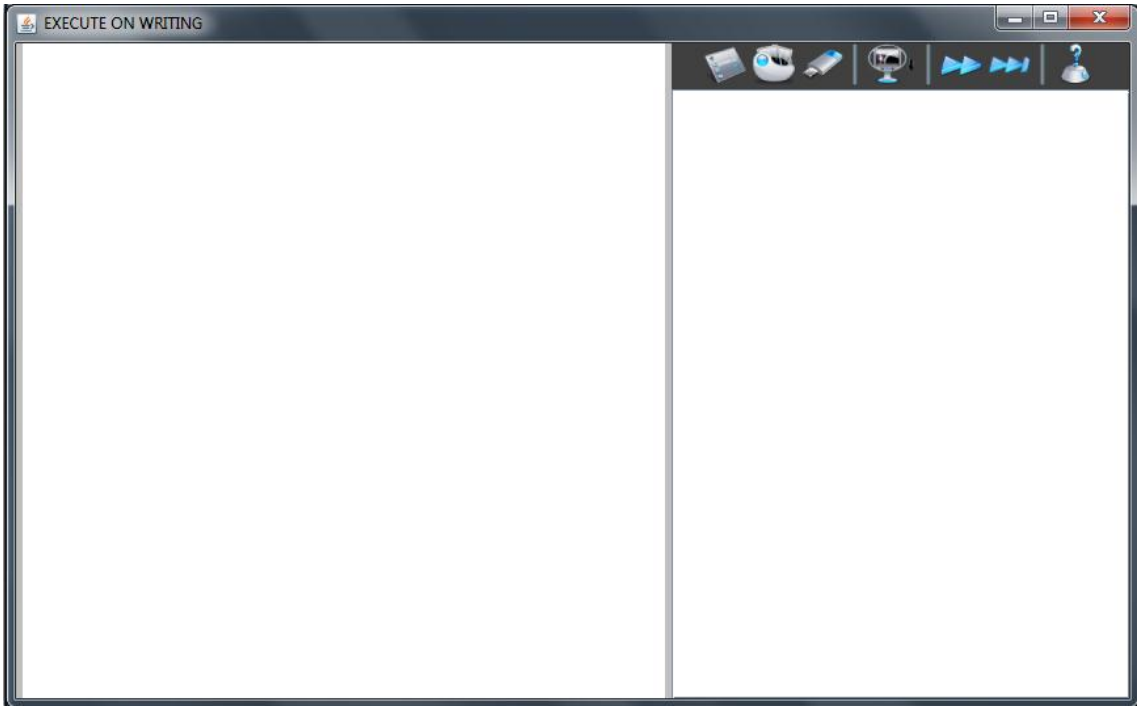

Figura 69. Aspecto inicial del modo Ejecución en Escritura

Una vez iniciado este modo, lo que debemos hacer es generar una secuencia de instrucciones que definan un árbol. Tenemos dos opciones para crear estas secuencias, escribiendo directamente en el panel derecho o cargando un fichero de texto que contenga la secuencia. La sintaxis y el repertorio de instrucciones disponibles es el siguiente:

1. Cada instrucción debe terminar en ';'
2. Las secuencias comenzarán por una de las siguientes palabras: "integer", "char", "string": que indicarán el tipo de los elementos del árbol que se va a generar.
3. La instrucción que permite añadir elementos es "add valor", con minúsculas.
4. Para eliminar un elemento utilizamos "del valor", también con minúsculas.
5. Usaremos la instrucción "draw" cuando queramos colocar breakpoints en zonas de código. Este flag permite generar sólo la secuencia de instrucciones que hay hasta llegar a él. Si queremos generar todas, no lo usaremos.
6. Para marcar el fin de una secuencia, utilizamos la instrucción "end"

Tras crear nuestra secuencia, visualizaremos el árbol resultante pulsando en el icono "Execute 'till draw" o "Execute 'till end". Como en todas las secciones, podremos guardar nuestros ficheros de secuencias en formato .txt y cargarlos cuando creamos conveniente.

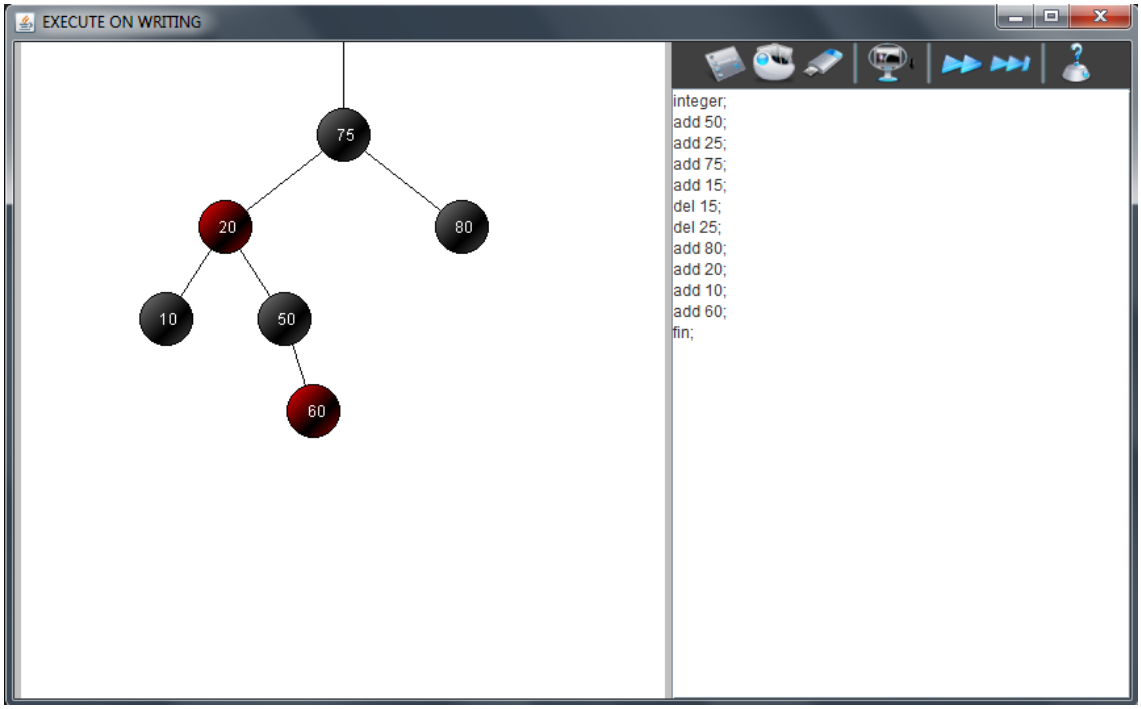

Figura 70. Aspecto tras introducir código y ejecutarlo

Para salir de este modo, sólo tenemos que cerrar la ventana y volveremos a la pizarra.

Si queremos cambiar de sección, sólo tenemos que pulsar sobre los iconos de la derecha para trasladarnos a AVL o cualquiera de los Tests.

3.4.3. Test AVL y Test Rojinegro

Desde la interfaz inicial, y una vez ya nos hayamos registrado como alumno, pulsamos el botón “Test AVL” o “Test Rojinegros”, botones tercero y cuarto de la columna derecha respectivamente.

Figura 71. Ventana inicial tras loguearse

En nuestra prueba elegiremos "TestAVL" como ejemplo. Lo primero que se nos mostrará será una pequeña pantalla con un JComboBox en la cual se mostrarán todos los tests disponibles hasta el momento sobre la estructura arborescente elegida.

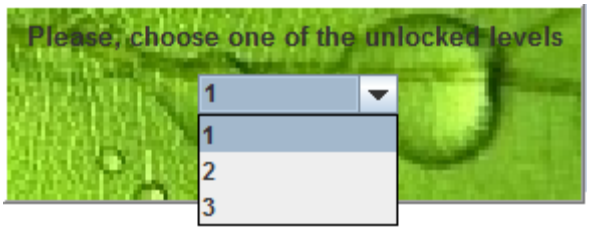

Figura 72. Selector de Test

Elegiremos una de las opciones y se mostrará una nueva interfaz en donde aparecerán las imágenes del cuestionario elegido y un apartado de creación de árboles a la izquierda para realizar pequeñas pruebas que ayuden a la realización de los tests por parte del alumno.

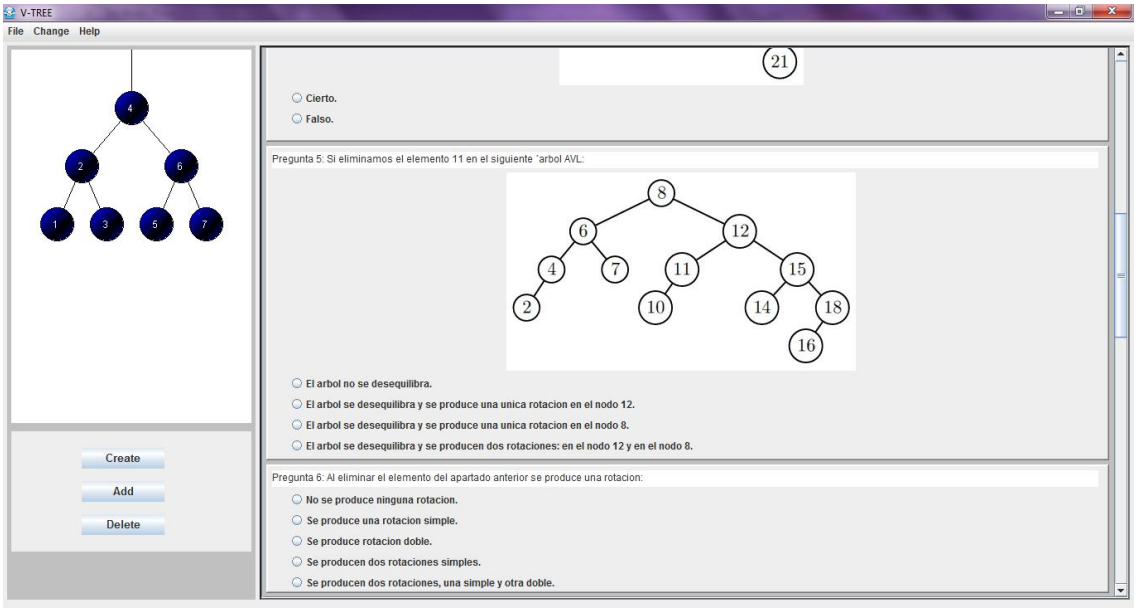

Figura 73. Aspecto de la ventana en modo test para alumnos

Al final del cuestionario aparecerá un apartado para poder mandar comentarios al profesor acerca de las preguntas en caso de que se desee recalcar algún aspecto del cuestionario, dudas, etc.

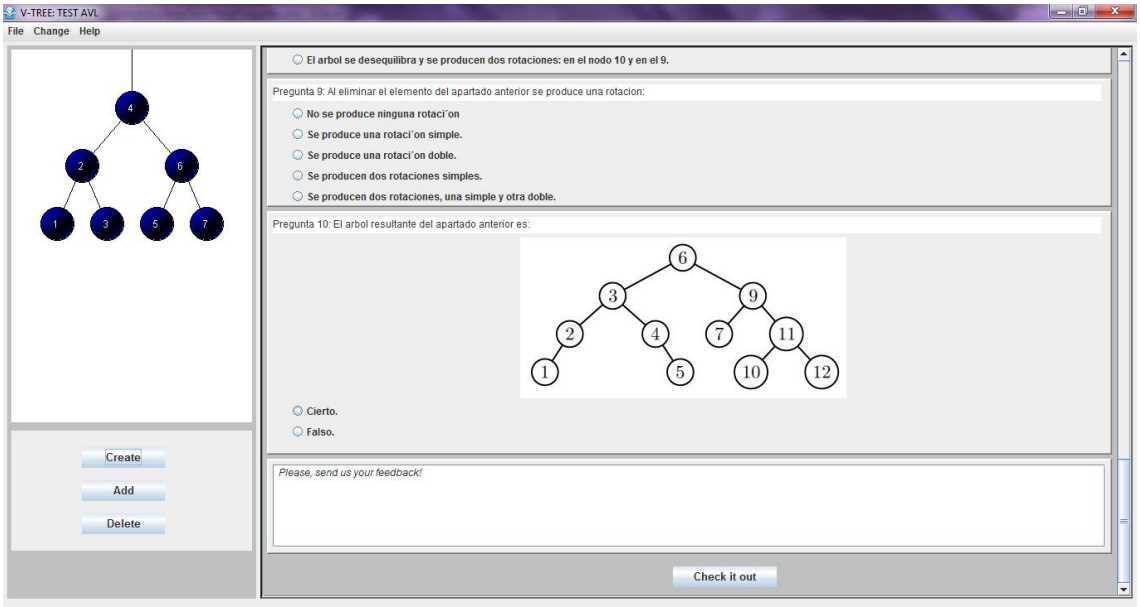

Figura 74. Muestra de la casilla de comentarios

Tendremos que contestar las diez preguntas del cuestionario para poder continuar. En caso contrario aparecerá un mensaje de advertencia en el que nos recordará que hay que rellenarlo por completo. No es necesario dejar ningún comentario si no se desea.

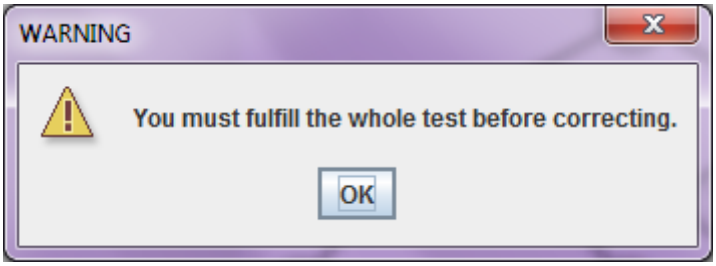

Figura 75. Aviso de que se debe rellenar todo el test

Una vez hayamos realizado la acción anterior, pulsaremos el botón "Check it out" para comprobar si nuestras respuestas han sido las correctas o no. A la derecha de cada opción tomada podrá aparecer un tick verde si ha sido la correcta; en caso contrario estará señalada con una cruz roja y al lado de la opción adecuada estará la señal de correcto. En el caso de que hubiéramos escrito algún comentario, este se almacenará en un fichero que recogerá todas las opiniones vertidas en dichos espacios para mandarlos al correo del profesor al cierre de la aplicación.

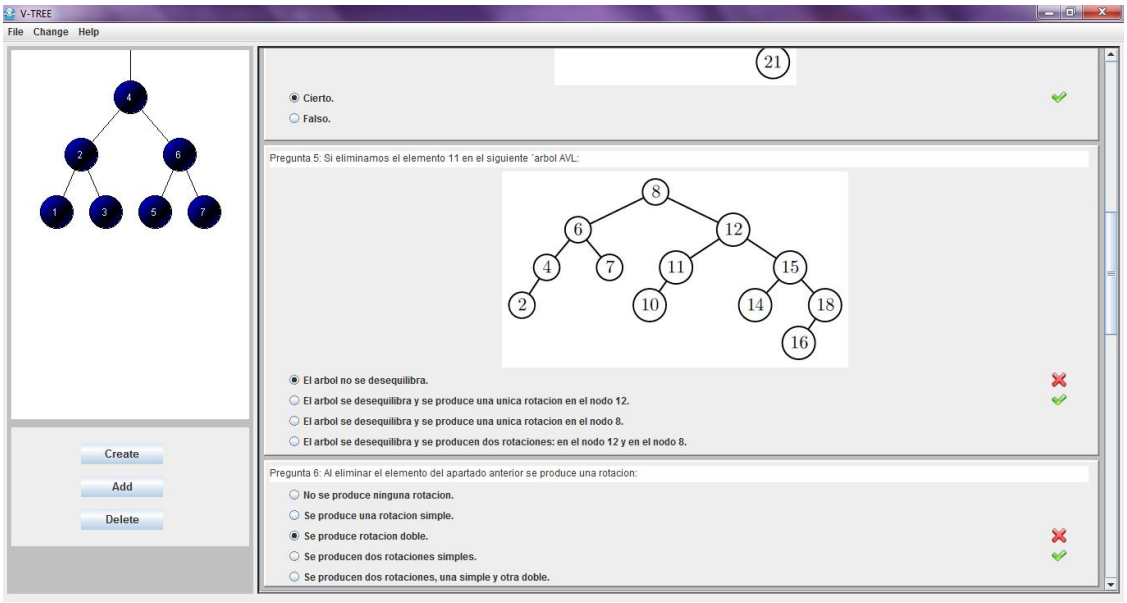

Figura 76. Aspecto del modo Test para alumnos tras la corrección

Si el alumno quiere realizar otro test de la misma estructura arborescente, ya sea repetir este mismo o uno diferente, acudirá en la barra de herramientas a "File"-> "New". Se cerrará la ventana del cuestionario actual y volverá a aparecer la pequeña ventana de selección de nivel. Una vez seleccionado de nuevo el deseado, se abrirá de nuevo la ventana con el test requerido y con el panel de manipulación de árboles del margen izquierdo limpio.

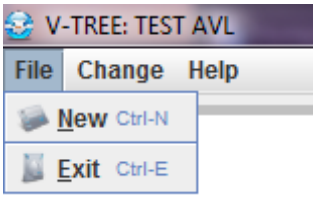

Figura 77. Contenido de pestaña "File"

Si nuestro objetivo es cambiar de opción, en “Change” en la barra de herramientas se podrá seleccionar la opción adecuada.

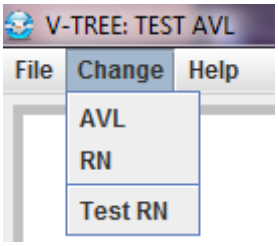

Figura 78. Contenido de pestaña "Change"

Así nuestra ventana actual se cerrará y se abrirá la correspondiente al apartado escogido. Si cerramos la aplicación y el fichero en el que se almacenan los comentarios como hemos dicho anteriormente, no estuviera vacío, estos se enviarán automáticamente al correo del profesor.

3.5. Logros y Limitaciones

V-Tree no está ni mucho menos en su fase final. Los objetivos que se han logrado cumplir a lo largo de este desarrollo son muchos y muy positivos. El entorno gráfico de representación de estructuras de datos arborescentes es una parte muy lograda del proyecto y en la que se ha invertido la mayoría del tiempo de desarrollo. Además, se ha conseguido ofrecer una interfaz de usuario muy sencilla e intuitiva que recoge todo lo necesario para procurar su correcta utilización y el correcto aprendizaje del campo que nos atañe.

Las diferentes formas que ofrece **V-Tree** para crear estructuras, manipularlas, guardarlas y cargarlas otorgan mucha flexibilidad y dinamismo a la herramienta.

En cuanto a la creación y realización de los tests de evaluación, **V-Tree** también ha conseguido crear un entorno intuitivo y ligero para los usuarios, tanto profesores como alumnos. La posibilidad de crear imágenes de las estructuras generadas y la opción de generar ficheros de secuencias de instrucciones que marcan los puntos críticos de un conjunto de inserciones y eliminaciones sobre un árbol, aporta mucha versatilidad a la hora de construir

preguntas de tests, que pueden incluir imágenes aclaratorias sobre una pregunta. Con sólo comenzar a crear un árbol en la pizarra podemos obtener infinidad de preguntas distintas.

Queda pendiente para trabajo futuro, aprovechar la potencia de la herramienta para generar preguntas automáticas con sus correspondientes soluciones.

Por el contrario, por diversos obstáculos encontrados durante el desarrollo de la herramienta, no se ha conseguido terminar de implementar el Tutor inteligente que controle la evolución del alumno. Esta parte se dejará como trabajo futuro.

Tampoco se ha conseguido probar la aplicación en su entorno natural, el Campus Virtual de la UCM por lo que quedará también pendiente la implantación del mismo y estudiar su utilidad.

3.6. Trabajo futuro

V-Tree es una herramienta con un gran potencial que se puede ver incrementado gracias a algunas de las siguientes ampliaciones. Su mejora lo convertiría en la herramienta perfecta no sólo para el aprendizaje, sino también para su comercialización.

- Acceso online a la aplicación: Esta opción debe ser la primera en efectuarse. Su cumplimiento conllevaría una optimización primordial en el proyecto. No solamente se podría acceder, facilitando la conexión desde cualquier ordenador con internet, sin ocupar espacio físico en la CPU. Si no que evitaría al profesor dar de alta a sus estudiantes antes de que los mismos realizaran la descarga de la aplicación. Habría una única base de registros, de test e imágenes que se actualizarían dinámicamente y podrían modificarse, eliminarse siendo los cambios visibles al instante.

- Opción de cambio de contraseña: Actualmente cada alumno sólo puede registrarse en la aplicación con la contraseña que le fue enviada al e-mail de la facultad. Esto les permitiría obtener una consigna personal más fácil de recordar. Esta mejora sería posible siempre y cuando la anterior se haya realizado con éxito.

- Optimización de determinadas clases de interfaces: Algunas de las clases podrían ser extendidas a partir de otras, o derivadas. Evitaríamos repetición de código y podríamos crear objetos que podrían ser reutilizados continuamente.

- Desbloqueo de los niveles de test según aprobados: Que solamente se desbloqueasen los tests de niveles superiores al actual en caso de que este se apruebe. Para esta opción es muy importante que la herramienta pueda estar online, dado que en caso contrario al cambiar el alumno de computadora tendrá que enfrentarse a todos los tests desde el inicio al no tener sus logros almacenados.

- Crear una base de datos sobre los alumnos: La creación de una base de datos en donde almacenáramos ciertos datos como, además del usuario y la contraseña, el número de rotaciones que ha realizado el alumno, el número de tests que ha hecho y aprobado, hasta qué nivel ha llegado, etc.

- Modificación de los tests: Si queremos modificar un test no podemos. Sólo se puede eliminar y volver a crearlo.

- Encriptación de los tests: Almacenar los tests tras haberlos encriptado, ahorrando espacio y evitando que puedan ser leídos y modificados por terceras personas en su propio beneficio si carecen de los permisos adecuados.

- Modificación de una pregunta y su actualización en los tests en los que aparece: Actualmente, si se modifica una pregunta de la base de datos y esta aparece en uno o más tests, no se actualizará en ellos. Entendemos que la mayoría de las veces si se modifica una pregunta es porque contiene un error; y buscarla a través de los tests para corregirla no debería hacerse a mano.

- Externalización de "Strings" para la personalización del idioma: La externalización de ciertas cadenas de texto como las mostradas por los botones, mensajes de aviso, etc, permitirían su traducción a otros idiomas, facilitando la labor de estudio de estudiantes de erasmus o su implantación en otras universidades o sitios de formación.

- Mejora de la documentación "Javadoc" del proyecto: Una descripción más detallada de la función de cada método, clase, objeto, etc para que cualquier futura mejora, reparación pueda ser realizada fácilmente por un programador sin conocimiento de la herramienta.

- Ampliación del proyecto: Podría extenderse para recopilar estructuras de datos pertenecientes a la clase de "Estructura de Datos y de la Información" como los albergados en "Metodología y Teoría de la Programación". La gran recopilación de estructuras, junto a las amplias posibilidades que ofrece de por sí esta herramienta, crearían una aplicación difícilmente emulable y novedosa en el sector.

- Generación de preguntas aleatorias: En la parte del Test sería interesante desarrollar una extensión que creara preguntas aleatorias basadas en un árbol seleccionado o bien construido también aleatoriamente. Mejoraría la comprensión del alumno sobre la estructura de datos al enfrentarse a preguntas que no ha respondido anteriormente, convirtiendose la herramienta en dinámica y pudiendo dar lugar a nuevas dudas del alumno que no se habría planteado de otra forma.

- Manual interactivo de instrucción: Inclusión de un botón o menú en cada una de las partes de la aplicación, tanto para profesor como alumno, que explicase con ejemplos interactivos como manejar los distintos componentes. Por ejemplo, para crear un árbol, señalaría el botón "Create" explicándote que puede escoger entre "integer", "char" y "String", pero no continuaría hasta que se hubiera seleccionado. Una vez completado el paso, avanzaría al siguiente, indicando que se podría "Add" un elemento, y señalando el correspondiente botón. Mostraría como hacer rotaciones de cada clase, como crear tests para los alumnos...Facilitaría la familiarización con el entorno, al convertirlo en algo ameno y visual sin necesidad de leer un manual.

- Mejora de las estadísticas: Aplicación de métodos estadísticos más complejos y orientados a las expectativas del profesor. Podrían ser implementados un mínino de dos, contando con una característica en la parte del profesor que le permitiera elegir cuál de ellos aplicar en cada momento.

- Inclusión de gráficos para las estadísticas: Como parte de su aplicación, generarían unos gráficos para ver la evolución general de la clase, el uso que le dan los alumnos, etc, así como el avance de cada alumno individualmente.

- Visualización de los resultados de las estadísticas por parte del profesor desde dentro de la aplicación: En la parte del test correspondiente al profesor, habría una opción que permitiría ver las estadísticas realizadas hasta el momento tanto en general como en particular. Los resultados podrían consultarse globalmente; pero también podríamos seleccionar una alumno y ver hasta que nivel ha desbloqueado en los tests, sus resultados, su evolución en el tiempo.

- Consulta desde la aplicación, de los comentarios que se han mandado al correo del profesor: Entendiendo que los comentarios no son dudas urgentes, una solución para evitar la masificación de estos posibles mensajes en el correo del docente sería almacenarlos en la aplicación. Una vez el profesor los leyera y los comprobase en caso de ser necesario, podrían ser borrados o almacenados según sus decisiones.

- Correo directo con el profesor para dudas concretas: Contamos con una opción de mandar comentarios desde los tests pero no implica que sean respondidas, tan sólo leídas por el profesor por ser consejos, fallos,etc. Pero, si la duda surgiera en la parte práctica de los árboles tendríamos que acceder al e-mail de la facultad, buscar el correo del profesor, quizás ajuntar una captura de pantalla y explicarlo. Fácilmente los alumnos podrían dejarlo para otro momento y perder la oportunidad de resolver sus dudas. Accediendo directamente desde la aplicación ahorrarían tiempo, pudiendo adjuntar la imagen del propio árbol de la pestaña en la que se encuentran. Por supuesto, también podrían consultar sobre la parte del test.

- Implementación avanzada del Tutor Inteligente que dirija la evolución de un alumno.

- Aplicación de la estructura de la herramienta en otros campos de enseñanza: Quizás en algunos estudios, como en química, una modificación de esta aplicación sería viable a la hora de realizar formulaciones y test sobre las mismas.

2.7. Bibliografía

[1] Sistemas Informáticos 2005 – 2006. Facultad de Informática UCM.

[2] Sistemas Informáticos 2007 – 2008. Facultad de Informática UCM.

[3] R. del Vado. Rotaciones en un Árbol AVL.

[4] Recursos teóricos y prácticos obtenidos de la red:

- Capturas de pantalla en Java
  http://javalangnullpointer.wordpress.com/2007/03/30/capturar-la-pantalla-con-java-screen-capture/
- Código JTree con JCheckNodes como nodos
  http://www.java2s.com/Code/Java/Swing-JFC/CheckBoxNodeTreeSample.htm
- Generador de claves aleatorias:
  http://www.redeszone.net/2012/02/14/generador-de-claves-wpa-y-wpa2-en-java
- JavaMail:
  http://www.chuidiang.com/java/herramientas/javamail/enviar-correo-javamail.php
  http://www.chuidiang.com/java/herramientas/javamail/enviar-adjuntos-javamail.php

[5] Plataforma Moodle. Campus Virtual UCM

Bibliografía adicional:

- I.Pita. Sistema multiusuario para la gestión universitaria vía web, incluyendo el desarrollo de un interfaz para el diseño de aplicaciones de datos visuales: Funcionamiento de árboles rojinegros. Curso 2005-2006.
- G. Urdaneta. Árboles Rojo-Negros. Año 2010
- R. del Vado. Árboles equilibrados: AVL
- R. del Vado. Rotaciones en un Árbol AVL.
- R. del Vado. Ejemplos de Rotaciones en Árboles RN.
- R. del Vado, P. Fernández, S. Muñoz, A. Murillo. Un Tutor Inteligente para la Visualización de Métodos Algorítmicos y Estructuras de Datos. Curso 2008-2009.